数算cheating paper

一、基础语法

类型转变

函数	描述
[int(x ,base]) x -- 字符串或数字，base -- 进制数，默认十进制	将x转换为一个整数
[long(x ,base])	将x转换为一个长整数
float(x)	将x转换到一个浮点数
[complex(real ,imag])	创建一个复数
str(x)	将对象 x 转换为字符串
repr(x)	将对象 x 转换为表达式字符串
eval(str)	用来计算在字符串中的有效Python表达式,并返回一个对象
tuple(s)	将序列 s 转换为一个元组
list(s)	将序列 s 转换为一个列表
set(s)	转换为可变集合
dict(d)	创建一个字典。d 必须是一个序列 (key,value)元组。
frozenset(s)	转换为不可变集合
chr(x)	将一个整数转换为一个字符
unichr(x)	将一个整数转换为Unicode字符
ord(x)	将一个字符转换为它的整数值
hex(x)	将一个整数转换为一个十六进制字符串
oct(x)	将一个整数转换为一个八进制字符串

```
>>> int('12',16)          # 如果是带参数base的话，12要以字符串的形式进行输入，12 为 16进制
18
```

运算符

+	加 - 两个对象相加	a + b 输出结果 30
-	减 - 得到负数或是一个数减去另一个数	a - b 输出结果 -10

+	**加 - 两个对象相加**	**a + b 输出结果 30**
*	乘 - 两个数相乘或是返回一个被重复若干次的字符串	a * b 输出结果 200
/	除 - x除以y	b / a 输出结果 2
%	取模 - 返回除法的余数	b % a 输出结果 0
**	幂 - 返回x的y次幂	a**b 为10的20次方， 输出结果 100000000000000000000
//	取整除 - 返回商的整数部分（**向下取整**）	`>>> 9//2 4 >>> -9//2 -5`

运算符	**描述**	**实例**
==	等于 - 比较对象是否相等	(a == b) 返回 False。
!=	不等于 - 比较两个对象是否不相等	(a != b) 返回 True。
>	大于 - 返回x是否大于y	(a > b) 返回 False。
<	小于 - 返回x是否小于y。所有比较运算符返回1表示真，返回0表示假。这分别与特殊的变量 True 和 False 等价。	(a < b) 返回 True。
>=	大于等于 - 返回x是否大于等于y。	(a >= b) 返回 False。
<=	小于等于 - 返回x是否小于等于y。	(a <= b) 返回 True。

运算符	**描述**	**实例**
&	按位与运算符：参与运算的两个值,如果两个相应位都为1,则该位的结果为1,否则为0	(a & b) 输出结果 12 ， 二进制解释: 0000 1100
|	按位或运算符：只要对应的二个二进位有一个为1时，结果位就为1。	(a | b) 输出结果 61 ， 二进制解释: 0011 1101
^	按位异或运算符：当两对应的二进位相异时，结果为1	(a ^ b) 输出结果 49 ， 二进制解释: 0011 0001
~	按位取反运算符：对数据的每个二进制位取反,即把1变为0,把0变为1 。**~x** 类似于 **-x-1**	(~a) 输出结果 -61 ， 二进制解释: 1100 0011，在一个有符号二进制数的补码形式。

运算符	描述	实例
<<	左移动运算符：运算数的各二进位全部左移若干位，由 **<<** 右边的数字指定了移动的位数，高位丢弃，低位补0。	a << 2 输出结果 240 ，二进制解释：1111 0000
>>	右移动运算符：把">>"左边的运算数的各二进位全部右移若干位，**>>** 右边的数字指定了移动的位数	a >> 2 输出结果 15 ，二进制解释：0000 1111

运算符	逻辑表达式	描述	实例
and	x and y	布尔"与" - 如果 x 为 False，x and y 返回 False，否则它返回 y 的计算值。	(a and b) 返回 20。
or	x or y	布尔"或" - 如果 x 是非 0，它返回 x 的计算值，否则它返回 y 的计算值。	(a or b) 返回 10。
not	not x	布尔"非" - 如果 x 为 True，返回 False 。如果 x 为 False，它返回 True。	not(a and b) 返回 False

运算符	描述	实例
in	如果在指定的序列中找到值返回 True，否则返回 False。	x 在 y 序列中 , 如果 x 在 y 序列中返回 True。
not in	如果在指定的序列中没有找到值返回 True，否则返回 False。	x 不在 y 序列中 , 如果 x 不在 y 序列中返回 True。

运算符	描述	实例
is	is 是判断两个标识符是不是引用自一个对象	**x is y**, 类似 **id(x) == id(y)** , 如果引用的是同一个对象则返回 True，否则返回 False
is not	is not 是判断两个标识符是不是引用自不同对象	**x is not y** ， 类似 **id(a) != id(b)**。如果引用的不是同一个对象则返回结果 True，否则返回 False。

以下表格列出了从最高到最低优先级的所有运算符：

运算符	描述
**	指数 (最高优先级)
~ + -	按位翻转, 一元加号和减号 (最后两个的方法名为 +@ 和 -@)
* / % //	乘，除，取模和取整除
+ -	加法减法

运算符	描述
>> <<	右移，左移运算符
&	位 'AND'
^ |	位运算符
<= < > >=	比较运算符
<> == !=	等于运算符
= %= /= //= -= += *= **=	赋值运算符
is is not	身份运算符
in not in	成员运算符
not and or	逻辑运算符

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
'tan', 'tanh', 'tau', 'trunc']
>>>math.sin?
>>>Signature: math.sin(x, /)
Docstring: Return the sine of x (measured in radians).
Type:      builtin_function_or_method
```

函数	返回值 (描述)
abs(x)	返回数字的绝对值，如abs(-10) 返回 10
ceil(x)	返回数字的上入整数，如math.ceil(4.1) 返回 5
cmp(x, y)	如果 x < y 返回 -1, 如果 x == y 返回 0, 如果 x > y 返回 1
exp(x)	返回e的x次幂(ex),如math.exp(1) 返回2.718281828459045
fabs(x)	以浮点数形式返回数字的绝对值，如math.fabs(-10) 返回10.0
floor(x)	返回数字的下舍整数，如math.floor(4.9)返回 4
log(x)	如math.log(math.e)返回1.0,math.log(100,10)返回2.0
log10(x)	返回以10为基数的x的对数，如math.log10(100)返回 2.0
max(x1, x2,...)	返回给定参数的最大值，参数可以为序列。
min(x1, x2,...)	返回给定参数的最小值，参数可以为序列。

函数	返回值 (描述)
modf(x)	返回x的整数部分与小数部分，两部分的数值符号与x相同，整数部分以浮点型表示。
pow(x, y)	x**y 运算后的值。
[round(x ,n])	返回浮点数x的四舍五入值，如给出n值，则代表舍入到小数点后的位数。
sqrt(x)	返回数字x的平方根

字符串

操作符	描述	实例
+	字符串连接	>>>a + b 'HelloPython'
*	重复输出字符串	>>>a * 2 'HelloHello'
[]	通过索引获取字符串中字符	>>>a[1] 'e'
[:]	截取字符串中的一部分	>>>a[1:4] 'ell'
in	成员运算符 - 如果字符串中包含给定的字符返回 True	>>>"H" in a True
not in	成员运算符 - 如果字符串中不包含给定的字符返回 True	>>>"M" not in a True
r/R	原始字符串 - 原始字符串：所有的字符串都是直接按照字面的意思来使用，没有转义特殊或不能打印的字符。 原始字符串除在字符串的第一个引号前加上字母"r"（可以大小写）以外，与普通字符串有着几乎完全相同的语法。	>>>print r'\n' \n >>> print R'\n' \n
%	格式字符串	请看下一章节

```
print "My name is %s and weight is %d kg!" % ('Zara', 21)
My name is Zara and weight is 21 kg!
```

格式化操作符辅助指令:

符 号	描述
%c	格式化字符及其ASCII码
%s	格式化字符串
%d	格式化整数
%u	格式化无符号整型
%o	格式化无符号八进制数

符 号	描述
%x	格式化无符号十六进制数
%X	格式化无符号十六进制数（大写）
%f	格式化浮点数字，可指定小数点后的精度
%e	用科学计数法格式化浮点数
%E	作用同%e，用科学计数法格式化浮点数
%g	%f和%e的简写
%G	%F 和 %E 的简写
%p	用十六进制数格式化变量的地址

符号	功能
*	定义宽度或者小数点精度
-	用做左对齐
+	在正数前面显示加号(+)
	在正数前面显示空格
#	在八进制数前面显示零('0')，在十六进制前面显示'0x'或者'0X'(取决于用的是'x'还是'X')
0	显示的数字前面填充'0'而不是默认的空格
%	'%%'输出一个单一的'%'
(var)	映射变量(字典参数)
m.n.	m 是显示的最小总宽度,n 是小数点后的位数(如果可用的话)

Python 的字符串内建函数

询问操作：str.lower?

方法	描述
string.capitalize()	把字符串的第一个字符大写
string.center(width)	返回一个原字符串居中,并使用空格填充至长度 width 的新字符串
string.count(str, beg=0, end=len(string))	返回 str 在 string 里面出现的次数，如果 beg 或者 end 指定则返回指定范围内 str 出现的次数
string.decode(encoding='UTF-8', errors='strict')	以 encoding 指定的编码格式解码 string，如果出错默认报一个 ValueError 的 异 常 ， 除非 errors 指 定 的 是 'ignore' 或 者'replace'

方法	描述
string.encode(encoding='UTF-8', errors='strict')	以 encoding 指定的编码格式编码 string，如果出错默认报一个ValueError 的异常，除非 errors 指定的是'ignore'或者'replace'
string.endswith(obj, beg=0, end=len(string))	检查字符串是否以 obj 结束，如果beg 或者 end 指定则检查指定的范围内是否以 obj 结束，如果是，返回 True,否则返回 False.
string.expandtabs(tabsize=8)	把字符串 string 中的 tab 符号转为空格，tab 符号默认的空格数是 8。
string.find(str, beg=0, end=len(string))	检测 str 是否包含在 string 中，如果 beg 和 end 指定范围，则检查是否包含在指定范围内，如果是返回开始的索引值，否则返回-1
string.format()	格式化字符串
string.index(str, beg=0, end=len(string))	跟find()方法一样，只不过如果str不在 string中会报一个异常.
string.isalnum()	如果 string 至少有一个字符并且所有字符都是字母或数字则返回 True,否则返回 False
string.isalpha()	如果 string 至少有一个字符并且所有字符都是字母则返回 True,否则返回 False
string.isdecimal()	如果 string 只包含十进制数字则返回 True 否则返回 False.
string.isdigit()	如果 string 只包含数字则返回 True 否则返回 False.
string.islower()	如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是小写，则返回 True，否则返回 False
string.isnumeric()	如果 string 中只包含数字字符，则返回 True，否则返回 False
string.isspace()	如果 string 中只包含空格，则返回 True，否则返回 False.
string.istitle()	如果 string 是标题化的(见 title())则返回 True，否则返回 False
string.isupper()	如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是大写，则返回 True，否则返回 False
string.join(seq)	以 string 作为分隔符，将 seq 中所有的元素(的字符串表示)合并为一个新的字符串
string.ljust(width)	返回一个原字符串左对齐,并使用空格填充至长度 width 的新字符串
string.lower()	转换 string 中所有大写字符为小写.

方法	描述
string.lstrip()	截掉 string 左边的空格
string.maketrans(intab, outtab)	maketrans() 方法用于创建字符映射的转换表，对于接受两个参数的最简单的调用方式，第一个参数是字符串，表示需要转换的字符，第二个参数也是字符串表示转换的目标。
max(str)	返回字符串 *str* 中最大的字母。
min(str)	返回字符串 *str* 中最小的字母。
string.partition(str)	有点像 find()和 split()的结合体,从 str 出现的第一个位置起,把字符串 string 分成一个 3 元素的元组(string_pre_str,str,string_post_str),如果 string 中不包含str 则 string_pre_str == string.
string.replace(str1, str2, num=string.count(str1))	把 string 中的 str1 替换成 str2,如果 num 指定，则替换不超过 num 次.
string.rfind(str, beg=0,end=len(string))	类似于 find() 函数，返回字符串最后一次出现的位置，如果没有匹配项则返回 -1。
string.rindex(str, beg=0,end=len(string))	类似于 index()，不过是返回最后一个匹配到的子字符串的索引号。
string.rjust(width)	返回一个原字符串右对齐,并使用空格填充至长度 width 的新字符串
string.rpartition(str)	类似于 partition()函数,不过是从右边开始查找
string.rstrip()	删除 string 字符串末尾的空格.
string.split(str="", num=string.count(str))	以 str 为分隔符切片 string，如果 num 有指定值，则仅分隔 **num+1** 个子字符串
[string.splitlines(keepends])	按照行('\r', '\r\n', '\n')分隔，返回一个包含各行作为元素的列表，如果参数 keepends 为 False，不包含换行符，如果为 True，则保留换行符。
string.startswith(obj, beg=0,end=len(string))	检查字符串是否是以 obj 开头，是则返回 True，否则返回 False。如果beg 和 end 指定值，则在指定范围内检查.
[string.strip(obj])	在 string 上执行 lstrip()和 rstrip()
string.swapcase()	翻转 string 中的大小写
string.title()	返回"标题化"的 string,就是说所有单词都是以大写开始，其余字母均为小写(见 istitle())
string.translate(str, del="")	根据 str 给出的表(包含 256 个字符)转换 string 的字符,要过滤掉的字符放到 del 参数中
string.upper()	转换 string 中的小写字母为大写
string.zfill(width)	返回长度为 width 的字符串，原字符串 string 右对齐，前面填充0

```
symbol = "-";
seq = ("a", "b", "c"); # 字符串序列
print symbol.join( seq );
```

列表

Python 表达式	结果	描述
len([1, 2, 3])	3	长度
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	组合
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	重复
3 in [1, 2, 3]	True	元素是否存在于列表中
for x in [1, 2, 3]: print x,	1 2 3	迭代

序号	函数
1	cmp(list1, list2) 比较两个列表的元素
2	len(list) 列表元素个数
3	max(list) 返回列表元素最大值
4	min(list) 返回列表元素最小值
5	list(seq) 将元组转换为列表

序号	方法
1	list.append(obj) 在列表末尾添加新的对象
2	list.count(obj) 统计某个元素在列表中出现的次数
3	list.extend(seq) 在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）
4	list.index(obj) 从列表中找出某个值第一个匹配项的索引位置
5	list.insert(index, obj) 将对象插入列表
6	[list.pop(index=-1]) 移除列表中的一个元素（默认最后一个元素），并且返回该元素的值
7	list.remove(obj) 移除列表中某个值的第一个匹配项
8	list.reverse() 反向列表中元素
9	list.sort(cmp=None, key=None, reverse=False) 对原列表进行排序

字典

```
dict = {}
dict['one'] = "This is one"
```

```
dict[2] = "This is two"

tinydict = {'name': 'runoob','code':6734, 'dept': 'sales'}

print dict['one']          # 输出键为'one' 的值
print dict[2]              # 输出键为 2 的值
print tinydict             # 输出完整的字典
print tinydict.keys()      # 输出所有键
print tinydict.values()    # 输出所有值

This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'runoob'}
['dept', 'code', 'name']
['sales', 6734, 'runoob']

>>>dict()                          # 创建空字典
{}
>>> dict(a='a', b='b', t='t')     # 传入关键字
{'a': 'a', 'b': 'b', 't': 't'}
>>> dict(zip(['one', 'two', 'three'], [1, 2, 3]))   # 映射函数方式来构造字典
{'three': 3, 'two': 2, 'one': 1}
>>> dict([('one', 1), ('two', 2), ('three', 3)])    # 可迭代对象方式来构造字典
{'three': 3, 'two': 2, 'one': 1}
>>>
```

```
d = {key1 : value1, key2 : value2 }

>>> tinydict = {'a': 1, 'b': 2, 'b': '3'}
>>> tinydict['b']
'3'

tinydict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

tinydict['Age'] = 8 # 更新
tinydict['School'] = "RUNOOB" # 添加

tinydict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

del tinydict['Name']  # 删除键是'Name'的条目
tinydict.clear()      # 清空字典所有条目
del tinydict          # 删除字典
```

序号	函数及描述
1	cmp(dict1, dict2) 比较两个字典元素。
2	len(dict) 计算字典元素个数，即键的总数。
3	str(dict) 输出字典可打印的字符串表示。
4	type(variable) 返回输入的变量类型，如果变量是字典就返回字典类型。

序号	函数及描述
1	dict.clear() 删除字典内所有元素
2	dict.copy() 返回一个字典的浅复制
3	[dict.fromkeys(seq, val]) 创建一个新字典，以序列 seq 中元素做字典的键，val 为字典所有键对应的初始值
4	dict.get(key, default=None) 返回指定键的值，如果值不在字典中返回default值
5	dict.has_key(key) 如果键在字典dict里返回true，否则返回false。Python3 不支持。
6	dict.items() 以列表返回可遍历的(键, 值) 元组数组
7	dict.keys() 以列表返回一个字典所有的键
8	dict.setdefault(key, default=None) 和get()类似, 但如果键不存在于字典中，将会添加键并将值设为default
9	dict.update(dict2) 把字典dict2的键/值对更新到dict里
10	dict.values() 以列表返回字典中的所有值
11	[pop(key,default]) 删除字典给定键 key 所对应的值，返回值为被删除的值。key值必须给出。 否则，返回default值。
12	popitem() 返回并删除字典中的最后一对键和值。

可更改(mutable)与不可更改(immutable)对象

在 python 中，strings, tuples, 和 numbers 是不可更改的对象，而 list,dict 等则是可以修改的对象。

- **不可变类型：**变量赋值 **a=5** 后再赋值 **a=10**，这里实际是新生成一个 int 值对象 10，再让 a 指向它，而 5 被丢弃，不是改变a的值，相当于新生成了a。
- **可变类型：**变量赋值 **la=[1,2,3,4]** 后再赋值 **la[2]=5** 则是将 list la 的第三个元素值更改，本身la没有动，只是其内部的一部分值被修改了。

python 函数的参数传递：

- **不可变类型：**类似 c++ 的值传递，如 整数、字符串、元组。如fun（a），传递的只是a的值，没有影响a对象本身。比如在 fun（a）内部修改 a 的值，只是修改另一个复制的对象，不会影响 a 本身。
- **可变类型：**类似 c++ 的引用传递，如 列表，字典。如 fun（la），则是将 la 真正的传过去，修改后fun外部的la也会受影响

```
 def ChangeInt( a ):
    a = 10

b = 2
ChangeInt(b)
print b # 结果是 2
```

```
def changeme( mylist ):
   "修改传入的列表"
   mylist.append([1,2,3,4])
   print "函数内取值: ", mylist
   return

# 调用changeme函数
mylist = [10,20,30]
changeme( mylist )
print "函数外取值: ", mylist

函数内取值:  [10, 20, 30, [1, 2, 3, 4]]
函数外取值:  [10, 20, 30, [1, 2, 3, 4]]
```

```
def printinfo( name, age ):
   "打印任何传入的字符串"
   print "Name: ", name
   print "Age ", age
   return
printinfo( age=50, name="miki" )

def printinfo( name, age = 35 ):
   "打印任何传入的字符串"
   print "Name: ", name
   print "Age ", age
   return
printinfo( age=50, name="miki" )
printinfo( name="miki" )

def printinfo( arg1, *vartuple ):
   "打印任何传入的参数"
   print "输出: "
   print arg1
   for var in vartuple:
      print var
   return
printinfo( 10 )
printinfo( 70, 60, 50 )

sum = lambda arg1, arg2: arg1 + arg2
print "相加后的值为 : ", sum( 10, 20 )
print "相加后的值为 : ", sum( 20, 20 )
```

异常名称	描述
BaseException	所有异常的基类
SystemExit	解释器请求退出
KeyboardInterrupt	用户中断执行(通常是输入^C)
Exception	常规错误的基类

异常名称	描述
StopIteration	迭代器没有更多的值
GeneratorExit	生成器(generator)发生异常来通知退出
StandardError	所有的内建标准异常的基类
ArithmeticError	所有数值计算错误的基类
FloatingPointError	浮点计算错误
OverflowError	数值运算超出最大限制
ZeroDivisionError	除(或取模)零 (所有数据类型)
AssertionError	断言语句失败
AttributeError	对象没有这个属性
EOFError	没有内建输入,到达EOF 标记
EnvironmentError	操作系统错误的基类
IOError	输入/输出操作失败
OSError	操作系统错误
WindowsError	系统调用失败
ImportError	导入模块/对象失败
LookupError	无效数据查询的基类
IndexError	序列中没有此索引(index)
KeyError	映射中没有这个键
MemoryError	内存溢出错误(对于Python 解释器不是致命的)
NameError	未声明/初始化对象 (没有属性)
UnboundLocalError	访问未初始化的本地变量
ReferenceError	弱引用(Weak reference)试图访问已经垃圾回收了的对象
RuntimeError	一般的运行时错误
NotImplementedError	尚未实现的方法
SyntaxError	Python 语法错误
IndentationError	缩进错误
TabError	Tab 和空格混用
SystemError	一般的解释器系统错误
TypeError	对类型无效的操作
ValueError	传入无效的参数

异常名称	描述
UnicodeError	Unicode 相关的错误
UnicodeDecodeError	Unicode 解码时的错误
UnicodeEncodeError	Unicode 编码时错误
UnicodeTranslateError	Unicode 转换时错误
Warning	警告的基类
DeprecationWarning	关于被弃用的特征的警告
FutureWarning	关于构造将来语义会有改变的警告
OverflowWarning	旧的关于自动提升为长整型(long)的警告
PendingDeprecationWarning	关于特性将会被废弃的警告
RuntimeWarning	可疑的运行时行为(runtime behavior)的警告
SyntaxWarning	可疑的语法的警告
UserWarning	用户代码生成的警告

```
b="+".join(s)
print(b)

print("Cube = %d, Triple = (%d,%d,%d)"%(result[i][0],result[i][1]\
                                        ,result[i][2],result[i][3]))
for i in range(n):
    print(*c[i],sep=' ')
print("%.1f"%(value))
result = "{:.6f}".format(s[0])
print(result)

    maze.append(['.' for i in range(m+2)])
    for i in range(n):
        s=['.']+[j for j in input()]+['.']
        maze.append(s)
    maze.append(['.' for i in range(m+2)])
b = bin(item)  # 2进制
o = oct(item)  # 8进制
h = hex(item)  # 16进制
```

2.4 集合

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# 并集
union_set = set1 | set2
print("并集:", union_set)  # 输出: {1, 2, 3, 4, 5}

# 交集
```

```
intersection_set = set1 & set2
print("交集:", intersection_set)  # 输出：{3}

# 差集
difference_set = set1 - set2
print("差集:", difference_set)  # 输出：{1, 2}

# 对称差集
symmetric_difference_set = set1 ^ set2
print("对称差集:", symmetric_difference_set)  # 输出：{1, 2, 4, 5}
```

import相关

```
# pylint: skip-file
import heapq
from collections import defaultdict
from collections import dequeue
import bisect
from functools import lru_cache
@lru_cache(maxsize=None)
import sys
sys.setrecursionlimit(1<<32)
import math
math.ceil()  # 函数进行向上取整
math.floor() # 函数进行向下取整。
math.isqrt() # 开方取整
exit()
```

```
from collections import Counter
# 创建一个包含多个重复元素的列表/字典
data = [1, 2, 3, 4, 1, 2, 3, 1, 2, 1]
# 使用Counter函数统计各个元素出现的次数
counter_result = Counter(data)
print(counter_result)
#输出
Counter({1: 4, 2: 3, 3: 2, 4: 1})
```

内置函数

```
sorted(iterable[, key[, reverse]]) # 返回值
list.sort([key[,reberse]])

print(*list)

lambda
aim_list = sorted(list, key = lambda o: o[1]) #举例
```

```
python itertools.product(range(2), repeat=6) 生成6元元组，01的全排列
可用于: for l in itertools.product(range(n), repeat=(m))
```

动规

最长上升子序列

```
import bisect
n = int(input())
*lis, = map(int, input().split())
dp = [1e9]*n
for i in lis:
    dp[bisect.bisect_left(dp, i)] = i
print(bisect.bisect_left(dp, 1e8))
```

背包问题

```
n,b=map(int, input().split())
price=[0]+[int(i) for i in input().split()]
weight=[0]+[int(i) for i in input().split()]
bag=[[0]*(b+1) for _ in range(n+1)]
for i in range(1,n+1):
    for j in range(1,b+1):
        if weight[i]<=j:
            bag[i][j]=max(price[i]+bag[i-1][j-weight[i]], bag[i-1][j])
        else:
            bag[i][j]=bag[i-1][j]
print(bag[-1][-1])
```

```
# 压缩矩阵/滚动数组 方法
N,B = map(int, input().split())
*p, = map(int, input().split())
*w, = map(int, input().split())

dp=[0]*(B+1)
for i in range(N):
    for j in range(B, w[i] - 1, -1):
        dp[j] = max(dp[j], dp[j-w[i]]+p[i])

print(dp[-1])
```

二进制	十进制	十六进制	字符/缩写	解释
00000000	0	00	NUL (NULL)	空字符
00000001	1	01	SOH (Start Of Headling)	标题开始
00000010	2	02	STX (Start Of Text)	正文开始
00000011	3	03	ETX (End Of Text)	正文结束
00000100	4	04	EOT (End Of Transmission)	传输结束
00000101	5	05	ENQ (Enquiry)	请求
00000110	6	06	ACK (Acknowledge)	回应/响应/收到通知
00000111	7	07	BEL (Bell)	响铃

二进制	十进制	十六进制	字符/缩写	解释
00001000	8	08	BS (Backspace)	退格
00001001	9	09	HT (Horizontal Tab)	水平制表符
00001010	10	0A	LF/NL(Line Feed/New Line)	换行键
00001011	11	0B	VT (Vertical Tab)	垂直制表符
00001100	12	0C	FF/NP (Form Feed/New Page)	换页键
00001101	13	0D	CR (Carriage Return)	回车键
00001110	14	0E	SO (Shift Out)	不用切换
00001111	15	0F	SI (Shift In)	启用切换
00010000	16	10	DLE (Data Link Escape)	数据链路转义
00010001	17	11	DC1/XON (Device Control 1/Transmission On)	设备控制1/传输开始
00010010	18	12	DC2 (Device Control 2)	设备控制2
00010011	19	13	DC3/XOFF (Device Control 3/Transmission Off)	设备控制3/传输中断
00010100	20	14	DC4 (Device Control 4)	设备控制4
00010101	21	15	NAK (Negative Acknowledge)	无响应/非正常响应/拒绝接收
00010110	22	16	SYN (Synchronous Idle)	同步空闲
00010111	23	17	ETB (End of Transmission Block)	传输块结束/块传输终止
00011000	24	18	CAN (Cancel)	取消
00011001	25	19	EM (End of Medium)	已到介质末端/介质存储已满/介质中断
00011010	26	1A	SUB (Substitute)	替补/替换
00011011	27	1B	ESC (Escape)	逃离/取消
00011100	28	1C	FS (File Separator)	文件分割符
00011101	29	1D	GS (Group Separator)	组分隔符/分组符
00011110	30	1E	RS (Record Separator)	记录分离符
00011111	31	1F	US (Unit Separator)	单元分隔符
00100000	32	20	(Space)	空格
00100001	33	21	!	

二进制	十进制	十六进制	字符/缩写	解释
00100010	34	22	"	
00100011	35	23	#	
00100100	36	24	$	
00100101	37	25	%	
00100110	38	26	&	
00100111	39	27	'	
00101000	40	28	(	
00101001	41	29	)	
00101010	42	2A	*	
00101011	43	2B	+	
00101100	44	2C	,	
00101101	45	2D	-	
00101110	46	2E	.	
00101111	47	2F	/	
00110000	48	30	0	
00110001	49	31	1	
00110010	50	32	2	
00110011	51	33	3	
00110100	52	34	4	
00110101	53	35	5	
00110110	54	36	6	
00110111	55	37	7	
00111000	56	38	8	
00111001	57	39	9	
00111010	58	3A	:	
00111011	59	3B	;	
00111100	60	3C	<	
00111101	61	3D	=	
00111110	62	3E	>	

二进制	十进制	十六进制	字符/缩写	解释
00111111	63	3F	?	
01000000	64	40	@	
01000001	65	41	A	
01000010	66	42	B	
01000011	67	43	C	
01000100	68	44	D	
01000101	69	45	E	
01000110	70	46	F	
01000111	71	47	G	
01001000	72	48	H	
01001001	73	49	I	
01001010	74	4A	J	
01001011	75	4B	K	
01001100	76	4C	L	
01001101	77	4D	M	
01001110	78	4E	N	
01001111	79	4F	O	
01010000	80	50	P	
01010001	81	51	Q	
01010010	82	52	R	
01010011	83	53	S	
01010100	84	54	T	
01010101	85	55	U	
01010110	86	56	V	
01010111	87	57	W	
01011000	88	58	X	
01011001	89	59	Y	
01011010	90	5A	Z	
01011011	91	5B	[	

二进制	十进制	十六进制	字符/缩写	解释
01011100	92	5C	\	
01011101	93	5D	]	
01011110	94	5E	^	
01011111	95	5F	_	
01100000	96	60	`	
01100001	97	61	a	
01100010	98	62	b	
01100011	99	63	c	
01100100	100	64	d	
01100101	101	65	e	
01100110	102	66	f	
01100111	103	67	g	
01101000	104	68	h	
01101001	105	69	i	
01101010	106	6A	j	
01101011	107	6B	k	
01101100	108	6C	l	
01101101	109	6D	m	
01101110	110	6E	n	
01101111	111	6F	o	
01110000	112	70	p	
01110001	113	71	q	
01110010	114	72	r	
01110011	115	73	s	
01110100	116	74	t	
01110101	117	75	u	
01110110	118	76	v	
01110111	119	77	w	
01111000	120	78	x	

二进制	十进制	十六进制	字符/缩写	解释
01111001	121	79	y	
01111010	122	7A	z	
01111011	123	7B	{	
01111100	124	7C	|	
01111101	125	7D	}	
01111110	126	7E	~	
01111111	127	7F	DEL (Delete)	删除

二、栈、队列

栈

先进后出，用list实现

```
stack=[]#创建栈
stack.append()#压入元素
stack.pop()#取出最后一个元素

class Stack:#类实现
    def __init__(self):
        self.items = []
    def is_empty(self):
        return self.items == []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    def peek(self):
        return self.items[len(self.items)-1]
    def size(self):
        return len(self.items)
```

单调栈

1. 单调栈简介

> **单调栈（Monotone Stack）**：一种特殊的栈。在栈的「先进后出」规则基础上，要求「从 **栈顶** 到 **栈底** 的元素是单调递增（或者单调递减）」。其中满足从栈顶到栈底的元素是单调递增的栈，叫做「单调递增栈」。满足从栈顶到栈底的元素是单调递减的栈，叫做「单调递减栈」。

注意：这里定义的顺序是从「栈顶」到「栈底」。有的文章里是反过来的。本文全文以「栈顶」到「栈底」的顺序为基准来描述单调栈。

1.1 单调递增栈

> **单调递增栈**：只有比栈顶元素小的元素才能直接进栈，否则需要先将栈中比当前元素小的元素出栈，再将当前元素入栈。
>
> 这样就保证了：栈中保留的都是比当前入栈元素大的值，并且从栈顶到栈底的元素值是单调递增的。

单调递增栈的入栈、出栈过程如下：

- 假设当前进栈元素为 `x`，如果 `x` 比栈顶元素小，则直接入栈。
- 否则从栈顶开始遍历栈中元素，把小于 `x` 或者等于 `x` 的元素弹出栈，直到遇到一个大于 `x` 的元素为止，然后再把 `x` 压入栈中。

1.2 单调递减栈

> **单调递减栈**：只有比栈顶元素大的元素才能直接进栈，否则需要先将栈中比当前元素大的元素出栈，再将当前元素入栈。
>
> 这样就保证了：栈中保留的都是比当前入栈元素小的值，并且从栈顶到栈底的元素值是单调递减的。

单调递减栈的入栈、出栈过程如下：

- 假设当前进栈元素为 `x`，如果 `x` 比栈顶元素大，则直接入栈。
- 否则从栈顶开始遍历栈中元素，把大于 `x` 或者等于 `x` 的元素弹出栈，直到遇到一个小于 `x` 的元素为止，然后再把 `x` 压入栈中。

2. 单调栈适用场景

单调栈可以在时间复杂度为 $O(n)$ 的情况下，求解出某个元素左边或者右边第一个比它大或者小的元素。

2.1 寻找左侧第一个比当前元素大的元素

- 从左到右遍历元素，构造单调递增栈（从栈顶到栈底递增）：
  - 一个元素左侧第一个比它大的元素就是将其「插入单调递增栈」时的栈顶元素。
  - 如果插入时的栈为空，则说明左侧不存在比当前元素大的元素。

2.2 寻找左侧第一个比当前元素小的元素

- 从左到右遍历元素，构造单调递减栈（从栈顶到栈底递减）：
  - 一个元素左侧第一个比它小的元素就是将其「插入单调递减栈」时的栈顶元素。
  - 如果插入时的栈为空，则说明左侧不存在比当前元素小的元素。

2.3 寻找右侧第一个比当前元素大的元素

- 从左到右遍历元素，构造单调递增栈（从栈顶到栈底递增）：
  - 一个元素右侧第一个比它大的元素就是将其「弹出单调递增栈」时即将插入的元素。
  - 如果该元素没有被弹出栈，则说明右侧不存在比当前元素大的元素。
- 从右到左遍历元素，构造单调递增栈（从栈顶到栈底递增）：
  - 一个元素右侧第一个比它大的元素就是将其「插入单调递增栈」时的栈顶元素。
  - 如果插入时的栈为空，则说明右侧不存在比当前元素大的元素。

2.4 寻找右侧第一个比当前元素小的元素

- 从左到右遍历元素，构造单调递减栈（从栈顶到栈底递减）：
  - 一个元素右侧第一个比它小的元素就是将其「弹出单调递减栈」时即将插入的元素。
  - 如果该元素没有被弹出栈，则说明右侧不存在比当前元素小的元素。
- 从右到左遍历元素，构造单调递减栈（从栈顶到栈底递减）：
  - 一个元素右侧第一个比它小的元素就是将其「插入单调递减栈」时的栈顶元素。
  - 如果插入时的栈为空，则说明右侧不存在比当前元素小的元素。

上边的分类解法有点绕口，可以简单记为以下条规则：

- 无论哪种题型，都建议从左到右遍历元素。
- 查找 **「比当前元素大的元素」** 就用 **单调递增栈**，查找 **「比当前元素小的元素」** 就用 **单调递减栈。**
- 从 **「左侧」** 查找就看 **「插入栈」** 时的栈顶元素，从 **「右侧」** 查找就看 **「弹出栈」** 时即将插入的元素。

3. 单调栈模板

以从左到右遍历元素为例，介绍一下构造单调递增栈和单调递减栈的模板。

3.1 单调递增栈模板

```
def monotoneIncreasingStack(nums):
    stack = []
    for num in nums:
        while stack and num >= stack[-1]:
            stack.pop()
        stack.append(num)
```

3.2 单调递减栈模板

```
def monotoneDecreasingStack(nums):
    stack = []
    for num in nums:
        while stack and num <= stack[-1]:
            stack.pop()
        stack.append(num)
```

28190: 奶牛排队

http://cs101.openjudge.cn/practice/28190/

奶牛在熊大妈的带领下排成了一条直队。显然，不同的奶牛身高不一定相同……

现在，奶牛们想知道，如果找出一些连续的奶牛，要求最左边的奶牛 A 是最矮的，最右边的 B 是最高的，且 B 高于 A 奶牛。中间如果存在奶牛，则身高不能和 A,B 奶牛相同。问这样的奶牛最多会有多少头?

从左到右给出奶牛的身高，请告诉它们符合条件的最多的奶牛数（答案可能是 0,2，但不会是 1）。

输入

第一行一个正整数 N，表示奶牛的头数。(2<=N<=10000)

接下来 N 行，每行一个正整数，从上到下表示从左到右奶牛的身高 hi (1<=hi<=50000000) 。

输出

一行一个整数，表示最多奶牛数。

样例输入

```
sample input1:
5
1
2
3
4
1

sample output1:
4

#取第 1 头到第 4 头奶牛，满足条件且为最多。
```

样例输出

```
sample input2:
10
15
15
2
13
7
4
11
5
11
12

sample output2:
5
```

提示

tags: monotonous-stack

来源

hhy, https://www.luogu.com.cn/problem/P6510

因为有单调栈的提示，所以优先思考单调栈的性质：能够找到（向左）一个最长的区间，其上面的值都比当前位置要小，于是当前位置就是这个区间的最大值——这算是想到了一半；

后一半就着重解决最左端最小值的性质，其实和前面是对称的想法，我要找到一个位置，这个位置必须要前面最长的区间内，且是区间最小值，这利用了单调栈中的另一个性质：单调栈内的元素，是单调的，意味着每个元素都是其到下一个元素之间的最小值——而最小值是具有传递性的，也就是每个元素是该位置当前遍历到的位置之间的最小值——恰好满足我们的要求。

于是我们只要找第一步中区间是否包含第二步中单调栈的元素即可（这里可以选择线性遍历或者二分，二分查找是因为单调栈是单调的），选择被包含元素中最左端的元素，就是以当前遍历到的位置为最大值的最长连续区间。

综上，我们只需要两个单调栈，一个是递增栈，一个是非增栈就好了。

```
# 熊江凯、元培学院
from bisect import bisect_right as bl
lis,q1,q2,ans=[int(input())for _ in range(int(input()))],[-1],[-1],0
for i in range(len(lis)):
    while len(q1)>1 and lis[q1[-1]]>=lis[i]:q1.pop()
    while len(q2)>1 and lis[q2[-1]]<lis[i]:q2.pop()
    id=bl(q1,q2[-1])
    if id<len(q1):ans=max(ans,i-q1[id]+1)
    q1.append(i)
    q2.append(i)
print(ans)
```

例题

03704: 括号匹配

stack, http://cs101.openjudge.cn/practice/03704

在某个字符串（长度不超过100）中有左括号、右括号和大小写字母；规定（与常见的算数式子一样）任何一个左括号都从内到外与在它右边且距离最近的右括号匹配。写一个程序，找到无法匹配的左括号和右括号，输出原来字符串，并在下一行标出不能匹配的括号。不能匹配的左括号用"$"标注，不能匹配的右括号用"?"标注.

输入

输入包括多组数据，每组数据一行，包含一个字符串，只包含左右括号和大小写字母，**字符串长度不超过100**
注意：cin.getline(str,100)最多只能输入99个字符!

输出

对每组输出数据，输出两行，第一行包含原始输入字符，第二行由"$","?"和空格组成，"$"和"?"表示与之对应的左括号和右括号不能匹配。

样例输入

```
((ABCD(x)
)(rttyy())sss)(
```

样例输出

```
((ABCD(x)
$$
)(rttyy())sss)(
?            ?$
```

```
# https://www.cnblogs.com/huashanqingzhu/p/6546598.html

lines = []
while True:
    try:
        lines.append(input())
    except EOFError:
        break

ans = []
for s in lines:
    stack = []
    Mark = []
    for i in range(len(s)):
        if s[i] == '(':
            stack.append(i)
            Mark += ' '
        elif s[i] == ')':
            if len(stack) == 0:
                Mark += '?'
            else:
                Mark += ' '
                stack.pop()
        else:
            Mark += ' '

    while(len(stack)):
        Mark[stack[-1]] = '$'
        stack.pop()

    print(s)
    print(''.join(map(str, Mark)))
```

24591: 中序表达式转后序表达式

http://cs101.openjudge.cn/practice/24591/

中序表达式是运算符放在两个数中间的表达式。乘、除运算优先级高于加减。可以用"()"来提升优先级 --- 就是小学生写的四则算术运算表达式。中序表达式可用如下方式递归定义：

1）一个数是一个中序表达式。该表达式的值就是数的值。

2. 若a是中序表达式，则"(a)"也是中序表达式(引号不算)，值为a的值。
3. 若a,b是中序表达式，c是运算符，则"acb"是中序表达式。"acb"的值是对a和b做c运算的结果，且a是左操作数，b是右操作数。

输入一个中序表达式，要求转换成一个后序表达式输出。

输入

第一行是整数n(n<100)。接下来n行，每行一个中序表达式，数和运算符之间没有空格，长度不超过700。

输出

对每个中序表达式，输出转成后序表达式后的结果。后序表达式的数之间、数和运算符之间用一个空格分开。

样例输入

```
3
7+8.3
3+4.5*(7+2)
(3)*((3+4)*(2+3.5)/(4+5))
```

样例输出

```
7 8.3 +
3 4.5 7 2 + * +
3 3 4 + 2 3.5 + * 4 5 + / *
```

来源: Guo wei

Shunting Yard 算法是一种用于将中缀表达式转换为后缀表达式的算法。它由荷兰计算机科学家 Edsger Dijkstra 在1960年代提出，用于解析和计算数学表达式。

Shunting Yard 算法的主要思想是使用两个栈（运算符栈和输出栈）来处理表达式的符号。算法按照运算符的优先级和结合性，将符号逐个处理并放置到正确的位置。最终，输出栈中的元素就是转换后的后缀表达式。

以下是 Shunting Yard 算法的基本步骤：

1. 初始化运算符栈和输出栈为空。
2. 从左到右遍历中缀表达式的每个符号。
    - 如果是操作数（数字），则将其添加到输出栈。
    - 如果是左括号，则将其推入运算符栈。
    - 如果是运算符：
        - 如果运算符的优先级大于运算符栈顶的运算符，或者运算符栈顶是左括号，则将当前运算符推入运算符栈。
        - 否则，将运算符栈顶的运算符弹出并添加到输出栈中，直到满足上述条件（或者运算符栈为空）。
        - 将当前运算符推入运算符栈。
    - 如果是右括号，则将运算符栈顶的运算符弹出并添加到输出栈中，直到遇到左括号。将左括号弹出但不添加到输出栈中。
3. 如果还有剩余的运算符在运算符栈中，将它们依次弹出并添加到输出栈中。
4. 输出栈中的元素就是转换后的后缀表达式。

```python
def infix_to_postfix(expression):
    precedence = {'+':1, '-':1, '*':2, '/':2}
    stack = []
    postfix = []
    number = ''

    for char in expression:
        if char.isnumeric() or char == '.':
```

```
            number += char
        else:
            if number:
                num = float(number)
                postfix.append(int(num) if num.is_integer() else num)
                number = ''
            if char in '+-*/':
                while stack and stack[-1] in '+-*/' and precedence[char] <=
precedence[stack[-1]]:
                    postfix.append(stack.pop())
                stack.append(char)
            elif char == '(':
                stack.append(char)
            elif char == ')':
                while stack and stack[-1] != '(':
                    postfix.append(stack.pop())
                stack.pop()

    if number:
        num = float(number)
        postfix.append(int(num) if num.is_integer() else num)

    while stack:
        postfix.append(stack.pop())

    return ' '.join(str(x) for x in postfix)

n = int(input())
for _ in range(n):
    expression = input()
    print(infix_to_postfix(expression))
```

02754: 八皇后

dfs and similar, http://cs101.openjudge.cn/practice/02754

描述：会下国际象棋的人都很清楚：皇后可以在横、竖、斜线上不限步数地吃掉其他棋子。如何将8个皇后放在棋盘上（有8 * 8个方格），使它们谁也不能被吃掉！这就是著名的八皇后问题。

对于某个满足要求的8皇后的摆放方法，定义一个皇后串a与之对应，即$a=b_1b_2...b_8~$,其中b_i为相应摆法中第i行皇后所处的列数。已经知道8皇后问题一共有92组解（即92个不同的皇后串）。

给出一个数b，要求输出第b个串。串的比较是这样的：皇后串x置于皇后串y之前，当且仅当将x视为整数时比y小。

八皇后是一个古老的经典问题：**如何在一张国际象棋的棋盘上，摆放8个皇后，使其任意两个皇后互相不受攻击。**该问题由一位德国**国际象棋排局家 Max Bezzel** 于 1848年提出。严格来说，那个年代，还没有“德国”这个国家，彼时称作“普鲁士”。1850年，**Franz Nauck** 给出了第一个解，并将其扩展成了“ **n 皇后** ”问题，即**在一张 n** x **n 的棋盘上，如何摆放 n 个皇后，使其两两互不攻击。**历史上，八皇后问题曾惊动过“数学王子”高斯(Gauss)，而且正是 Franz Nauck 写信找高斯请教的。

输入

第1行是测试数据的组数n，后面跟着n行输入。每组测试数据占1行，包括一个正整数b(1 ≤ b ≤ 92)

输出

输出有n行，每行输出对应一个输入。输出应是一个正整数，是对应于b的皇后串。

样例输入

```
2
1
92
```

样例输出

```
15863724
84136275
```

思路：用dfs算法，判断两个皇后不在同一斜线的方法是保证两个皇后所在坐标的行差不等于列差。

```
# 赵昱安 2200011450
answer = []

def Queen(s):
    for col in range(1, 9):
        for j in range(len(s)):
            if (str(col) == s[j] or # 两个皇后不能在同一列
                    abs(col - int(s[j])) == abs(len(s) - j)): # 两个皇后不能在同一斜
线
                break
        else:
            if len(s) == 7:
                answer.append(s + str(col))
            else:
                Queen(s + str(col))

Queen('')

n = int(input())
for _ in range(n):
    a = int(input())
    print(answer[a - 1])
```

先给出两个dfs回溯实现的八皇后，接着给出两个stack迭代实现的八皇后。

八皇后思路：回溯算法通过尝试不同的选择，逐步构建解决方案，并在达到某个条件时进行回溯，以找到所有的解决方案。从第一行第一列开始放置皇后，然后在每一行的不同列都放置，如果与前面不冲突就继续，有冲突则回到上一行继续下一个可能性。

```
def solve_n_queens(n):
    solutions = []  # 存储所有解决方案的列表
    queens = [-1] * n  # 存储每一行皇后所在的列数

    def backtrack(row):
        if row == n:  # 找到一个合法解决方案
            solutions.append(queens.copy())
```

```
        else:
            for col in range(n):
                if is_valid(row, col):  # 检查当前位置是否合法
                    queens[row] = col  # 在当前行放置皇后
                    backtrack(row + 1)  # 递归处理下一行
                    queens[row] = -1  # 回溯，撤销当前行的选择

    def is_valid(row, col):
        for r in range(row):
            if queens[r] == col or abs(row - r) == abs(col - queens[r]):
                return False
        return True

    backtrack(0)  # 从第一行开始回溯

    return solutions


# 获取第 b 个皇后串
def get_queen_string(b):
    solutions = solve_n_queens(8)
    if b > len(solutions):
        return None
    queen_string = ''.join(str(col + 1) for col in solutions[b - 1])
    return queen_string


test_cases = int(input())  # 输入的测试数据组数
for _ in range(test_cases):
    b = int(input())  # 输入的 b 值
    queen_string = get_queen_string(b)
    print(queen_string)
```

队列

双端队列

先进先出

```
from collections import deque
st = "abcd"
list1 = [0, 1, 2, 3]
dst = deque(st)
dlist1 = deque(list1)
dst.append(4)#deque(['a', 'b', 'c', 'd', 4])
dst.appendleft(4)#deque([4, 'a', 'b', 'c', 'd'])
dst.extend(ex)#deque(['a', 'b', 'c', 'd', 1, 'h', 3])
p = dst.pop()#d#deque(['a', 'b', 'c'])
p = dst.popleft()#a#deque(['b', 'c', 'd'])

st = "abbcd"
p = dst.count("b")#2
dst.insert(0, "chl")#deque(['chl', 'a', 'b', 'b', 'c', 'd'])
```

```
dst.rotate(1)#deque(['d', 'a', 'b', 'b', 'c'])
dst.clear()#deque([])
dst.remove("a")#deque(['b', 'b', 'c', 'd'])
```

02746: 约瑟夫问题

implementation, http://cs101.openjudge.cn/practice/02746

约瑟夫问题：有n只猴子，按顺时针方向围成一圈选大王（编号从1到n），从第1号开始报数，一直数到m，数到m的猴子退出圈外，剩下的猴子再接着从1开始报数。就这样，直到圈内只剩下一只猴子时，这个猴子就是猴王，编程求输入n，m后，输出最后猴王的编号。

输入

每行是用空格分开的两个整数，第一个是 n, 第二个是 m (0 < m,n <=300)。最后一行是：

0 0

输出

对于每行输入数据（最后一行除外），输出数据也是一行，即最后猴王的编号

样例输入

```
6 2
12 4
8 3
0 0
```

样例输出

```
5
1
7
```

说明：使用 队列queue 这种数据结构会方便。它有三种实现方式，我们最常用的 list 就支持，说明，https://www.geeksforgeeks.org/queue-in-python/

```
from collections import deque

# 先使用pop从列表中取出，如果不符合要求再append回列表，相当于构成了一个圈
def hot_potato(name_list, num):
    queue = deque()
    for name in name_list:
        queue.append(name)

    while len(queue) > 1:
        for i in range(num):
            queue.append(queue.popleft()) # O(1)
        queue.popleft()
    return queue.popleft()


while True:
    n, m = map(int, input().split())
    if {n,m} == {0}:
```

```
        break
    monkey = [i for i in range(1, n+1)]
    print(hot_potato(monkey, m-1))
```

优先队列（最小堆）

heapq.heapify(x): O(n)

heapq.heappush(heap, item): O(logn)
heapq.heappop(heap): O(logn)

```
import heapq
# 创建优先队列
pq = []
# 添加元素到优先队列
heapq.heappush(pq, (5, 'low priority task'))#从[]开始自动建立小根堆
# 弹出优先级最高的元素
print(heapq.heappop(pq))
#从列表开始建立
lst = [5, 2, 4, 1, 3]
heapq.heapify(lst)
#合并（需要merge之前就有序）
heap1 = [1, 3, 5]
heap2 = [2, 4, 6]
merged_heap = heapq.merge(heap1, heap2)
```

三、树

树的定义

定义一：树由节点及连接节点的边构成。树有以下属性：
❑ 有一个根节点；
❑ 除根节点外，其他每个节点都与其唯一的父节点相连；
❑ 从根节点到其他每个节点都有且仅有一条路径；
❑ 如果每个节点最多有两个子节点，我们就称这样的树为二叉树。

定义二：一棵树要么为空，要么由一个根节点和零棵或多棵子树构成，子树本身也是一棵树。每棵子树的根节点通过一条边连到父树的根节点。图3展示了树的递归定义。从树的递归定义可知，图中的树至少有4个节点，因为三角形代表的子树必定有一个根节点。这棵树或许有更多的节点，但必须更深入地查看子树后才能确定。

树的组成

节点 Node：节点是树的基础部分。
每个节点具有名称，或“键值”。节点还可以保存额外数据项，数据项根据不同的应用而变。

边 Edge：边是组成树的另一个基础部分。
每条边恰好连接两个节点，表示节点之间具有关联，边具有出入方向；
每个节点（除根节点）恰有一条来自另一节点的入边；
每个节点可以有零条/一条/多条连到其它节点的出边。如果加限制不能有 “多条边”，这里树结构就特殊化为线性表

根节 Root: 树中唯一没有入边的节点。

路径 Path：由边依次连接在一起的有序节点列表。比如，哺乳纲→食肉目→猫科→猫属→家猫就是一条路径。

子节点 Children：入边均来自于同一个节点的若干节点，称为这个节点的子节点。

父节点 Parent：一个节点是其所有出边连接节点的父节点。

兄弟节点 Sibling：具有同一父节点的节点之间为兄弟节点。

子树 Subtree：一个节点和其所有子孙节点，以及相关边的集合。

叶节点 Leaf Node：没有子节点的节点称为叶节点。

层级 Level：
从根节点开始到达一个节点的路径，所包含的边的数量，称为这个节点的层级。

高度Height：所有节点层级的最大值。

二叉树深度：从根节点到叶节点依次经过的节点形成的树的一条路径，最长路径的节点个数为树的深度。

特别注意：根据定义，**depth=height+1**

树的表示

24729: 括号嵌套树

http://cs101.openjudge.cn/practice/24729/

可以用括号嵌套的方式来表示一棵树。表示方法如下：

1. 如果一棵树只有一个结点，则该树就用一个大写字母表示，代表其根结点。
2. 如果一棵树有子树，则用“树根(子树1,子树2,...,子树n)”的形式表示。树根是一个大写字母，子树之间用逗号隔开，没有空格。子树都是用括号嵌套法表示的树。

给出一棵不超过26个结点的树的括号嵌套表示形式，请输出其前序遍历序列和后序遍历序列。

输入

一行，一棵树的括号嵌套表示形式

输出

两行。第一行是树的前序遍历序列，第二行是树的后序遍历序列

样例输入

```
A(B(E),C(F,G),D(H(I)))
```

样例输出

```
ABECFGDHI
EBFGCIHDA
```

来源：Guo Wei

下面两个代码。先给出用类表示node。

思路：对于括号嵌套树，使用stack记录进行操作中的父节点，node记录正在操作的节点。每当遇见一个字母，将其设为node，并存入stack父节点中；遇到'('，即对当前node准备添加子节点，将其append入stack中，node重新设为None；遇到')'，stack父节点操作完毕，将其弹出并作为操作中的节点node，不断重复建立树，同时最后返出的父节点为树的根root。

前序遍历和后序遍历只要弄清楚意思，用递归很好写，注意这道题并不是二叉树，需要遍历解析树。

```
class TreeNode:
    def __init__(self, value): #类似字典
        self.value = value
        self.children = []

def parse_tree(s):
    stack = []
    node = None
    for char in s:
        if char.isalpha():  # 如果是字母，创建新节点
            node = TreeNode(char)
            if stack:  # 如果栈不为空，把节点作为子节点加入到栈顶节点的子节点列表中
                stack[-1].children.append(node)
        elif char == '(':  # 遇到左括号，当前节点可能会有子节点
            if node:
                stack.append(node)  # 把当前节点推入栈中
                node = None
        elif char == ')':  # 遇到右括号，子节点列表结束
            if stack:
                node = stack.pop()  # 弹出当前节点
    return node  # 根节点


def preorder(node):
    output = [node.value]
    for child in node.children:
        output.extend(preorder(child))
    return ''.join(output)

def postorder(node):
    output = []
    for child in node.children:
        output.extend(postorder(child))
    output.append(node.value)
    return ''.join(output)

# 主程序
def main():
    s = input().strip()
    s = ''.join(s.split())  # 去掉所有空白字符
    root = parse_tree(s)  # 解析整棵树
    if root:
        print(preorder(root))  # 输出前序遍历序列
        print(postorder(root))  # 输出后序遍历序列
    else:
```

```
        print("input tree string error!")

if __name__ == "__main__":
    main()
```

树的遍历

原理：

前序：根、左子树、右子树（根左右）

中序：左子树、根、右子树（左根右）

后序：左子树、右子树、跟（左右根）

24750: 根据二叉树中后序序列建树

http://cs101.openjudge.cn/dsapre/24750/

假设二叉树的节点里包含一个大写字母，每个节点的字母都不同。

给定二叉树的中序遍历序列和后序遍历序列(长度均不超过26)，请输出该二叉树的前序遍历序列。

输入

2行，均为大写字母组成的字符串，表示一棵二叉树的中序遍历序列与后序遍历排列。

输出

表示二叉树的前序遍历序列。

样例输入

```
BADC
BDCA
```

样例输出

```
ABCD
```

来源

Lou Yuke

```
"""
后序遍历的最后一个元素是树的根节点。然后，在中序遍历序列中，根节点将左右子树分开。
可以通过这种方法找到左右子树的中序遍历序列。然后，使用递归地处理左右子树来构建整个树。
"""
def build_tree(inorder, postorder):
    if not inorder or not postorder:
        return []

    root_val = postorder[-1]
    root_index = inorder.index(root_val)

    left_inorder = inorder[:root_index]
    right_inorder = inorder[root_index + 1:]
```

```
    left_postorder = postorder[:len(left_inorder)]
    right_postorder = postorder[len(left_inorder):-1]

    root = [root_val]
    root.extend(build_tree(left_inorder, left_postorder))
    root.extend(build_tree(right_inorder, right_postorder))

    return root


def main():
    inorder = input().strip()
    postorder = input().strip()
    preorder = build_tree(inorder, postorder)
    print(''.join(preorder))


if __name__ == "__main__":
    main()
```

25140: 根据后序表达式建立队列表达式

http://cs101.openjudge.cn/practice/25140/

后序算术表达式可以通过栈来计算其值，做法就是从左到右扫描表达式，碰到操作数就入栈，碰到运算符，就取出栈顶的2个操作数做运算(先出栈的是第二个操作数，后出栈的是第一个)，并将运算结果压入栈中。最后栈里只剩下一个元素，就是表达式的值。

有一种算术表达式不妨叫做“队列表达式”，它的求值过程和后序表达式很像，只是将栈换成了队列：从左到右扫描表达式，碰到操作数就入队列，碰到运算符，就取出队头2个操作数做运算（先出队的是第2个操作数，后出队的是第1个），并将运算结果加入队列。最后队列里只剩下一个元素，就是表达式的值。

给定一个后序表达式，请转换成等价的队列表达式。例如，`3 4 + 6 5 * -`的等价队列表达式就是`5 6 4 3 * + -`。

输入

第一行是正整数n(n<100)。接下来是n行，每行一个由字母构成的字符串，长度不超过100,表示一个后序表达式，其中小写字母是操作数，大写字母是运算符。运算符都是需要2个操作数的。

输出

对每个后序表达式，输出其等价的队列表达式。

样例输入

```
2
xyPzwIM
abcABdefgCDEF
```

样例输出

```
wzyxIPM
gfCecbDdAaEBF
```

提示

建立起表达式树，按层次遍历表达式树的结果前后颠倒就得到队列表达式

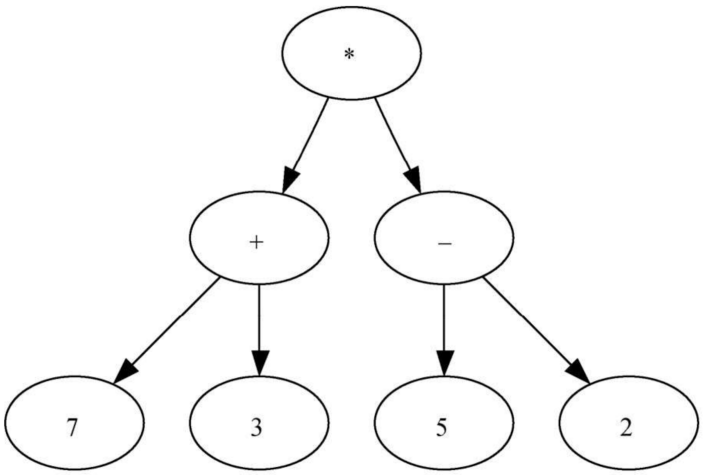
来源：Guo Wei modified from Ulm Local 2007

```
from collections import deque

class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def build_tree(postfix):
    stack = []
    for char in postfix:
        node = TreeNode(char)
        if char.isupper():
            node.right = stack.pop()
            node.left = stack.pop()
        stack.append(node)
    return stack[0]

def level_order_traversal(root):
    dq = [root]
    traversal = []
    while dq:
        node = dq.pop(0)
        traversal.append(node.value)
        if node.left:
            dq.append(node.left)
        if node.right:
            dq.append(node.right)
    return traversal

n = int(input().strip())
for _ in range(n):
    postfix = input().strip()
    root = build_tree(postfix)
    queue_expression = level_order_traversal(root)[::-1]
    print(''.join(queue_expression))
```

树的应用

① 解析树

我们可以将`( ( 7 + 3) * ( 5 - 2 ) )`这样的数学表达式表示成解析树



- **构建解析树**

  构建解析树的第一步是将表达式字符串拆分成标记列表。

  需要考虑4种标记：左括号、右括号、运算符和操作数。

  我们知道，左括号代表新表达式的起点，所以应该创建一棵对应该表达式的新树。

  反之，遇到右括号则意味着到达该表达式的终点。我们也知道，操作数既是叶子节点，也是其运算符的子节点。

  此外，每个运算符都有左右子节点。

  有了上述信息，便可以定义以下4条规则：

  (1) 如果当前标记是"("，就为当前节点添加一个左子节点，并下沉至该子节点；
  (2) 如果当前标记在列表 `['+', '-', '/', '*']` 中，就将当前节点的值设为当前标记对应的运算符；为当前节点添加一个右子节点，并下沉至该子节点；
  (3) 如果当前标记是数字，就将当前节点的值设为这个数并返回至父节点；
  (4) 如果当前标记是)，就跳到当前节点的父节点。

  以（3+（4*5））为例。

```
class Stack(object):
    def __init__(self):
        self.items = []
        self.stack_size = 0

    def isEmpty(self):
        return self.stack_size == 0

    def push(self, new_item):
        self.items.append(new_item)
        self.stack_size += 1

    def pop(self):
        self.stack_size -= 1
        return self.items.pop()

    def peek(self):
        return self.items[self.stack_size - 1]

    def size(self):
        return self.stack_size
```

```
class BinaryTree:
    def __init__(self, rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None

    def insertLeft(self, newNode):
        if self.leftChild == None:
            self.leftChild = BinaryTree(newNode)
        else:  # 已经存在左子节点。此时，插入一个节点，并将已有的左子节点降一层。
            t = BinaryTree(newNode)
            t.leftChild = self.leftChild
            self.leftChild = t

    def insertRight(self, newNode):
        if self.rightChild == None:
            self.rightChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.rightChild = self.rightChild
            self.rightChild = t

    def getRightChild(self):
        return self.rightChild

    def getLeftChild(self):
        return self.leftChild

    def setRootVal(self, obj):
        self.key = obj

    def getRootVal(self):
        return self.key

    def traversal(self, method="preorder"):
        if method == "preorder":
            print(self.key, end=" ")
        if self.leftChild != None:
            self.leftChild.traversal(method)
        if method == "inorder":
            print(self.key, end=" ")
        if self.rightChild != None:
            self.rightChild.traversal(method)
        if method == "postorder":
            print(self.key, end=" ")


def buildParseTree(fpexp):
    fplist = fpexp.split()
    pStack = Stack()
    eTree = BinaryTree('')
    pStack.push(eTree)
    currentTree = eTree

    for i in fplist:
        if i == '(':
```

```
            currentTree.insertLeft('')
            pStack.push(currentTree)
            currentTree = currentTree.getLeftChild()
        elif i not in '+-*/)':
            currentTree.setRootVal(int(i))
            parent = pStack.pop()
            currentTree = parent
        elif i in '+-*/':
            currentTree.setRootVal(i)
            currentTree.insertRight('')
            pStack.push(currentTree)
            currentTree = currentTree.getRightChild()
        elif i == ')':
            currentTree = pStack.pop()
        else:
            raise ValueError("Unknown Operator: " + i)
    return eTree


exp = "( ( 7 + 3 ) * ( 5 - 2 ) )"
pt = buildParseTree(exp)
for mode in ["preorder", "postorder", "inorder"]:
    pt.traversal(mode)
    print()

"""
* + 7 3 - 5 2
7 3 + 5 2 - *
7 + 3 * 5 - 2
"""


#代码清单6-14 后序求值
def postordereval(tree):
    opers = {'+':operator.add, '-':operator.sub,
             '*':operator.mul, '/':operator.truediv}
    res1 = None
    res2 = None
    if tree:
        res1 = postordereval(tree.getLeftChild())
        res2 = postordereval(tree.getRightChild())
        if res1 and res2:
            return opers[tree.getRootVal()](res1,res2)
        else:
            return tree.getRootVal()

print(postordereval(pt))
# 30

#代码清单6-16 中序还原完全括号表达式
def printexp(tree):
    sVal = ""
    if tree:
        sVal = '(' + printexp(tree.getLeftChild())
        sVal = sVal + str(tree.getRootVal())
        sVal = sVal + printexp(tree.getRightChild()) + ')'
```

```
    return sVal

print(printexp(pt))
# (((7)+3)*((5)-2))
```

解析树的构建、遍历可以和前、中、后续表达式联系起来。

② 二叉搜索树（Binary Searching Tree BST）

二叉搜索树（Binary Search Tree，BST），它是映射的另一种实现。我们感兴趣的不是元素在树中的确切位置，而是如何利用二叉树结构提供高效的搜索。

二叉搜索树依赖于这样一个性质：小于父节点的键都在左子树中，大于父节点的键则都在右子树中。我们称这个性质为二叉搜索性。

依赖于这一特性，二叉搜索树的中序遍历是有序的。通过这个办法可以实现树排序，平均时间复杂度为**$$nlogn$$**.但是当树的平衡性很差时时间复杂度会坍塌到$$O(n^2)$$.

③平衡二叉搜索树（AVLTree）

(a) 简介

当二叉搜索树不平衡时，get和put等操作的性能可能降到O(n)。本节将介绍一种特殊的二叉搜索树，它能**自动维持平衡。**这种树叫作 AVL树，以其发明者G. M. **A**delson-**V**elskii和E. M. **L**andis的姓氏命名。

实现：AVL树实现映射抽象数据类型的方式与普通的二叉搜索树一样，唯一的差别就是性能。实现AVL树时，要**记录每个节点的平衡因子。**我们**通过查看每个节点左右子树的高度来实现**这一点。更正式地说，我们**将平衡因子定义为左右子树的高度之差。**

$BalanceFactor = height (left SubTree) - height(right SubTree)$

根据上述定义，如果平衡因子大于零，我们称之为**左倾**；如果平衡因子小于零，就是**右倾**；如果平衡因子等于零，那么树就是**完全平衡**的。

为了实现AVL树并利用平衡树的优势，我们**将平衡因子为-1、0和1的树都定义为平衡树。**一旦某个节点的**平衡因子超出这个范围，我们就需要通过一个过程让树恢复平衡。**

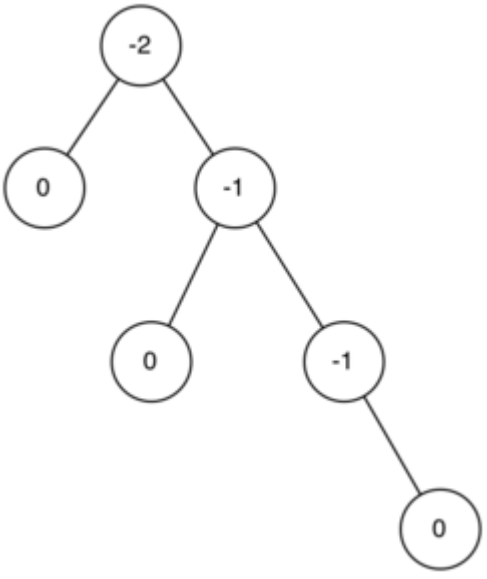


图1 带平衡因子的右倾树

假设现在已有一棵平衡二叉树，那么可以预见到，在往其中插入一个结点时，一定会有结点的平衡因子发生变化，此时可能会有结点的平衡因子的绝对值大于 1（这些平衡因子只可能是 2 或者 -2），这样以该结点为根结点的子树就是失衡的，需要进行调整。显然，只有在从根结点到该插入结点的路径上的结点才可能发生平衡因子变化，因此只需对这条路径上失衡的结点进行调整。

可以证明，**只要把最靠近插入结点的失衡结点调整到正常，路径上的所有结点就都会平衡。**

(b) 两种旋转操作

- **左旋**

> 如果需要进行再平衡，该怎么做呢？高效的再平衡是让AVL树发挥作用同时不损性能的关键。为了让AVL树恢复平衡，需要在树上进行一次或多次旋转。
>
> 要理解什么是旋转，来看一个简单的例子。考虑图2中左边的树。这棵树失衡了，平衡因子是-2。要让它恢复平衡，我们围绕以节点A为根节点的子树做一次左旋。

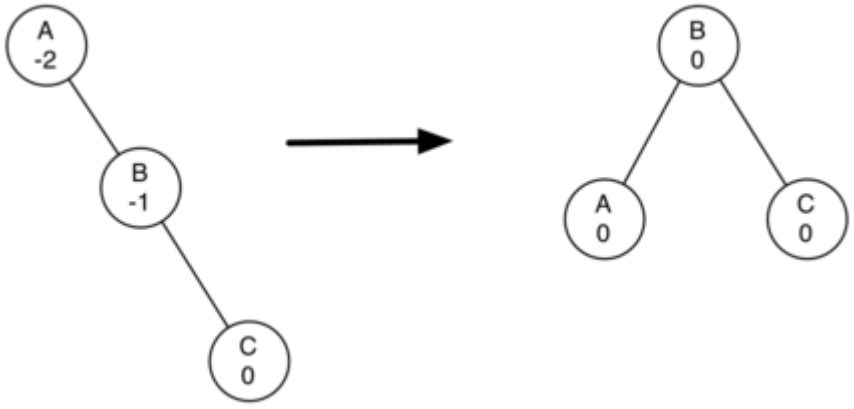


图2 通过左旋让失衡的树恢复平衡

> 本质上，左旋包括以下步骤。
>
> 1. **将右子节点（节点B）提升为子树的根节点。**
> 2. **将旧根节点（节点A）作为新根节点的左子节点。**
> 3. **如果新根节点（节点B）已经有一个左子节点，将其作为新左子节点（节点A）的右子节点。**
>
>   **注意，因为节点B之前是节点A的右子节点，所以此时节点A必然没有右子节点。因此，可以为它添加新的右子节点，而无须过多考虑。**

- **右旋**

> 我们来看一棵稍微复杂一点的树，并理解右旋过程。图4左边的是一棵左倾的树，根节点的平衡因子是2。右旋步骤如下。

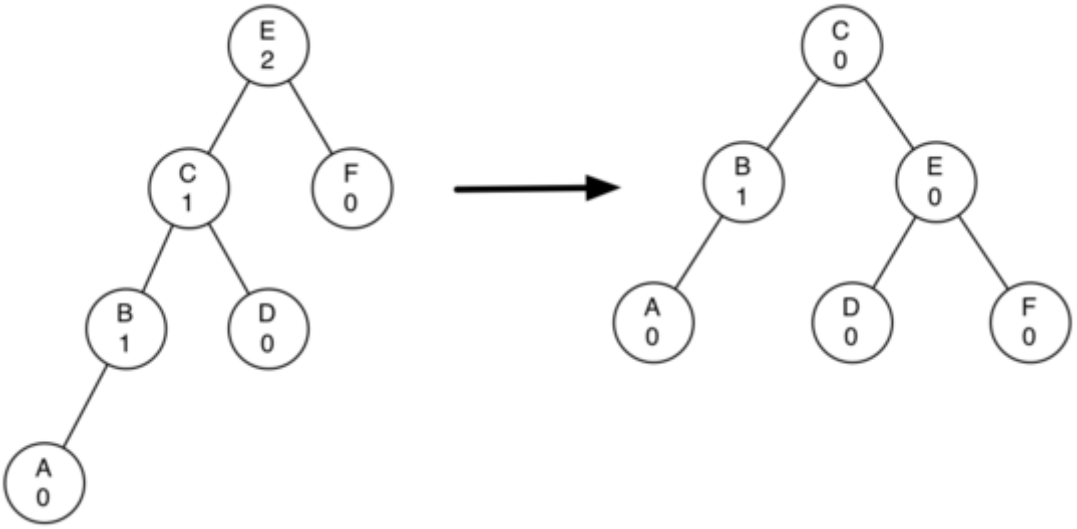


图3 通过右旋让失衡的树恢复平衡

1. **将左子节点（节点C）提升为子树的根节点。**
2. **将旧根节点（节点E）作为新根节点的右子节点。**
3. **如果新根节点（节点C）已经有一个右子节点（节点D），将其作为新右子节点（节点E）的左子节点。注意，因为节点C之前是节点E的左子节点，所以此时节点E必然没有左子节点。因此，可以为它添加新的左子节点，而无须过多考虑。**

(c) 四种树形以及对应调整办法

假设最靠近插入结点的失衡结点是 A，显然它的平衡因子只可能是 2 或者 -2。很容易发现这两种情况完全对称，因此主要讨论结点 A 的平衡因子是 2 的情形。

由于结点 A 的平衡因子是 2，因此左子树的高度比右子树大 2，于是以结点 A 为根结点的子树一定是图4的两种形态 LL 型与 LR 型之一（**注意：LL 和 LR 只表示树型，不是左右旋的意思**），其中☆、★、◇、◆是图中相应结点的 AVL 子树，结点 A、B、C 的权值满足 A > B > C。可以发现，**当结点 A 的左孩子的平衡因子是 1 时为 LL 型，是 -1 时为 LR 型。**那么，为什么结点 A 的左孩子的平衡因子只可能是 1 或者 -1，而不可能是 0 呢？这是因为这种情况无法由平衡二叉树插入一个结点得到。

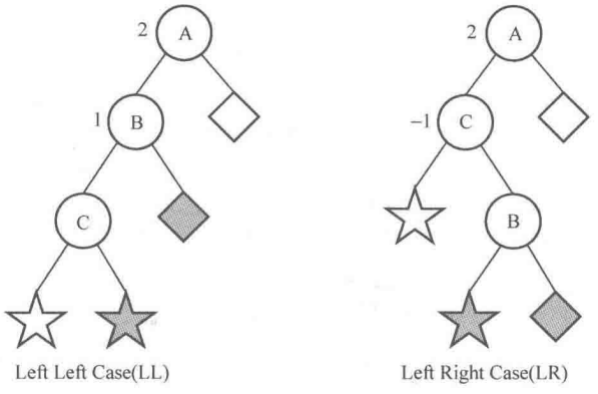


图4 树型之 LL 型与 LR 型（数字代表平衡因子）

补充说明，除了☆、★、◇、◆均为空树的情况以外，其他任何情况均满足：**在插入前，底层两棵子树的高度比另外两棵子树的高度小 1，且插入操作一定发生在底层两棵子树上。**例如对LL型来说，插入前子树的高度满足☆ = ★ = ◆-1 = ◇-1，而在☆或★中插入一个结点后导致☆或★的高度加 1，使得结点A不平衡。现在考虑怎样调整这两种树型，才能使树平衡。

> 先考虑 LL 型，可以把以 C 为根结点的子树看作一个整体，然后以结点 A 作为 root 进行右旋，便可以达到平衡，如图5 所示。

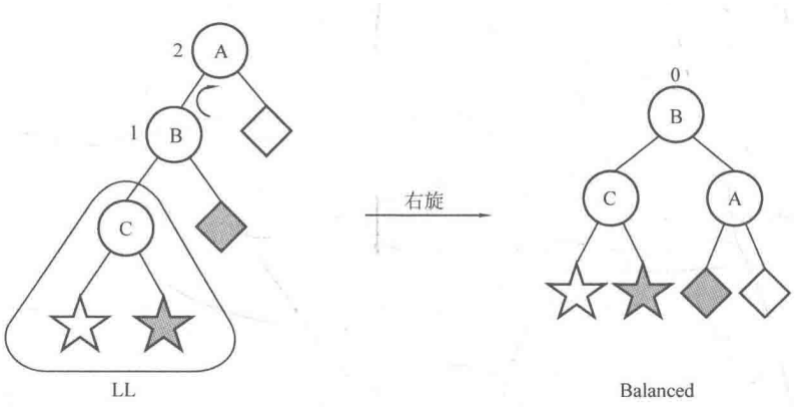


图5 LL 型调整示意图（数字代表平衡因子）

> 然后考虑 LR 型，可以先忽略结点 A，以结点 C 为root 进行左旋，就可以把情况转化为 LL 型，然后按上面 LL 型的做法进行一次右旋即可，如图6 所示。

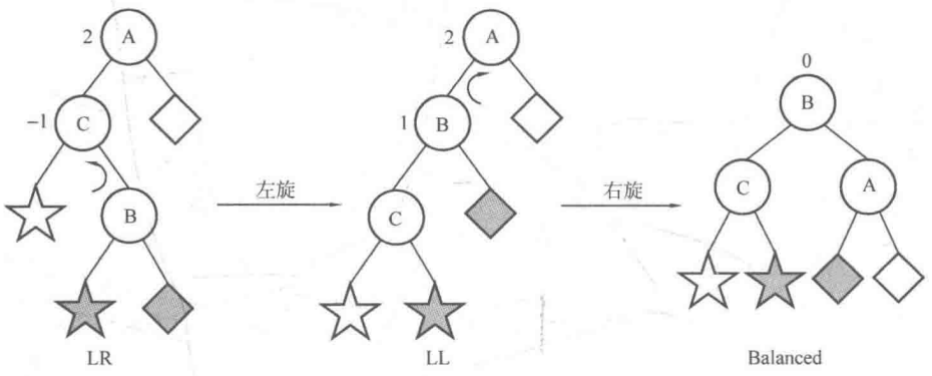


图6 LR型调整示意图（数字代表平衡因子）

至此,结点 A 的平衡因子是 2 的情况已经讨论清楚,下面简要说明平衡因子是 -2 的情况，显然两种情况是完全对称的。

由于结点 A 的平衡因子为 -2，因此右子树的高度比左子树大 2，于是以结点A为根结点的子树一定是图7的两种形态 RR 型与 RL 型之一。注意，由于和上面讨论的 LL 型和 LR 型对称，此处结点 A、B、C 的权值满足A < B < C。可以发现，**当结点 A 的右孩子的平衡因子是 -1 时为 RR 型，是1时为 RL 型。**

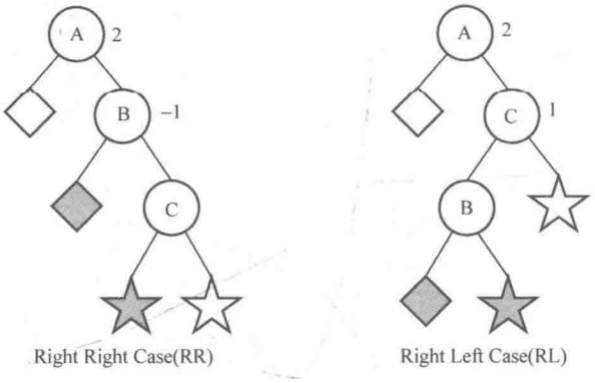


图7 树型之 RR型与RL型（数字代表平衡因子）

> 对 RR 型来说，可以把以 C 为根结点的子树看作一个整体，然后以结点 A 作为 root 进行左旋，便可以达到平衡，如图8 所示。

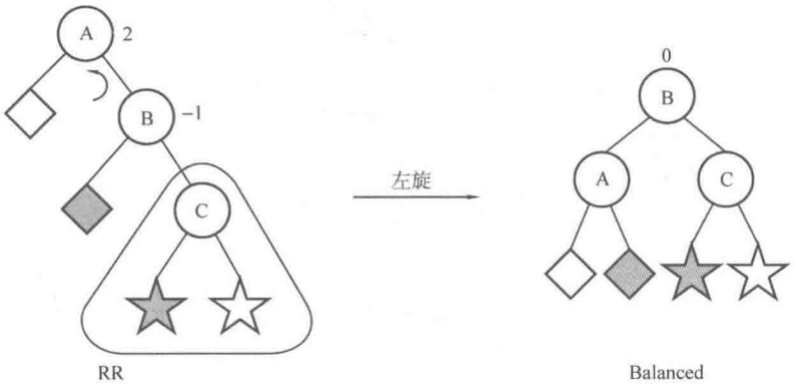


图8 RR 型调整示意图（数字代表平衡因子）

> 对 RL 型来说，可以先忽略结点 A，以结点 C 为 root 进行右旋，就可以把情况转化为 RR 然后按上面 RR 型的做法进行一次左旋即可，如图9 所示。

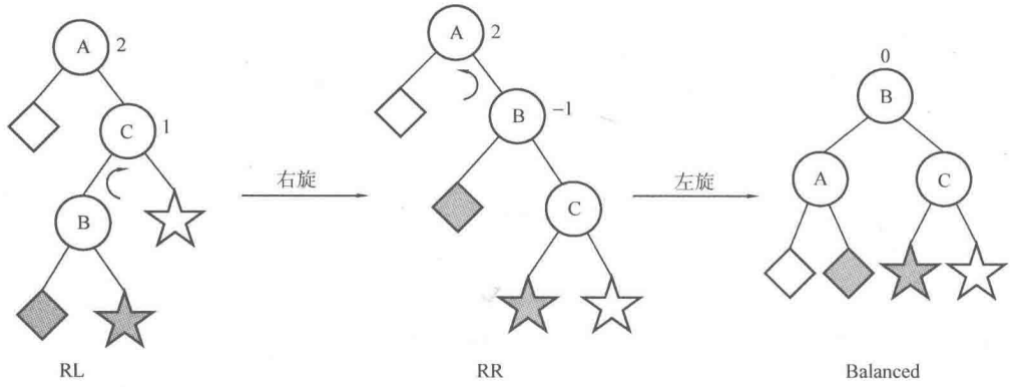


图9 RL型调整示意图（数字代表平衡因子）

至此，对**LL 型、LR 型、RR 型、RL型**的调整方法都已经讨论清楚。

通过维持树的平衡，可以保证get方法的时间复杂度为$O(log_2(n))$。但这会给put操作的性能带来多大影响呢？我们来看看put操作。因为新节点作为叶子节点插入，所以更新所有父节点的平衡因子最多需要$log_2(n)$次操作——每一层一次。如果树失衡了，恢复平衡最多需要旋转两次。每次旋转的时间复杂度是O(1)，所以put操作的时间复杂度仍然是$O(log_2(n))$。

至此，我们已经实现了一棵可用的AVL树。

(d) 建立AVLTree

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.height = 1
class AVL:
    def __init__(self):
        self.root = None
    def insert(self, value):
```

```
        if not self.root:
            self.root = Node(value)
        else:
            self.root = self._insert(value, self.root)

    def _insert(self, value, node):
        if not node:
            return Node(value)
        elif value < node.value:
            node.left = self._insert(value, node.left)
        else:
            node.right = self._insert(value, node.right)

        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))

        balance = self._get_balance(node)

        if balance > 1:
            if value < node.left.value: # 树形是 LL
                return self._rotate_right(node)
            else:   # 树形是 LR
                node.left = self._rotate_left(node.left)
                return self._rotate_right(node)

        if balance < -1:
            if value > node.right.value:    # 树形是 RR
                return self._rotate_left(node)
            else:   # 树形是 RL
                node.right = self._rotate_right(node.right)
                return self._rotate_left(node)

        return node

    def _get_height(self, node):
        if not node:
            return 0
        return node.height

    def _get_balance(self, node):
        if not node:
            return 0
        return self._get_height(node.left) - self._get_height(node.right)

    def _rotate_left(self, z):
        y = z.right
        T2 = y.left
        y.left = z
        z.right = T2
        z.height = 1 + max(self._get_height(z.left), self._get_height(z.right))
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        return y

    def _rotate_right(self, y):
        x = y.left
        T2 = x.right
```

```
        x.right = y
        y.left = T2
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        x.height = 1 + max(self._get_height(x.left), self._get_height(x.right))
        return x
```

④ Huffman算法（哈夫曼编码）

22161: 哈夫曼编码树

http://cs101.openjudge.cn/practice/22161/

根据字符使用频率(权值)生成一棵唯一的哈夫曼编码树。生成树时需要遵循以下规则以确保唯一性：

选取最小的两个节点合并时，节点比大小的规则是:

1. 权值小的节点算小。权值相同的两个节点，字符集里最小字符小的，算小。

例如 ({'c','k'},12) 和 ({'b','z'},12)，后者小。

2. 合并两个节点时，小的节点必须作为左子节点
3. 连接左子节点的边代表0,连接右子节点的边代表1

然后对输入的串进行编码或解码

```
import heapq
class HuffmanTreeNode:
    def __init__(self,weight,char=None):
        self.weight=weight
        self.char=char
        self.left=None
        self.right=None

    def __lt__(self,other):
        if self.weight==other.weight:
            return self.char<other.char
        return self.weight<other.weight

def BuildHuffmanTree(characters):
    heap=[HuffmanTreeNode(weight,char) for char,weight in characters.items()]
    heapq.heapify(heap)
    while len(heap)>1:
        left=heapq.heappop(heap)
        right=heapq.heappop(heap)
        merged=HuffmanTreeNode(left.weight+right.weight)
        merged.left=left
        merged.right=right
        heapq.heappush(heap,merged)
    root=heapq.heappop(heap)
    return root

def encode_huffman_tree(root):
    codes={}
    def traverse(node,code):
        if node.char:
```

```
            codes[node.char]=code
        else:
            traverse(node.left,code+'0')
            traverse(node.right,code+'1')
    traverse(root,"")
    return codes

def huffman_encoding(codes,string):
    encoded=""
    for char in string:
        encoded+=codes[char]
    return encoded

def huffman_decoding(root,encoded_string):
    decoded=""
    node=root
    for bit in encoded_string:
        if bit=='0':
            node=node.left
        else:
            node=node.right
        if node.char:
            decoded+=node.char
            node=root
    return decoded
```

⑤ 并查集（Disjoint Set）

```
class DisJointSet():
    def __init__(self,n):
        self.father={}
        self.rank={}
        for i in range(n):
            self.father[i]=i
            self.rank[i]=0
    def find(self,x):
        if self.father[x]!=x:
            self.father[x]=self.find(self.father[x])
        return self.father[x]
    def union(self,x,y):
        rootx=self.find(x)
        rooty=self.find(y)
        if rootx!=rooty:
            if self.rank[rootx]>self.rank[rooty]:
                self.father[rooty]=rootx
            elif self.rank[rootx]<self.rank[rooty]:
                self.father[rootx]=rooty
            else:
                self.father[rootx]=rooty
                self.rank[rooty]+=1
```

6.Trie

1. **插入（Insert）**：

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True
```

2. **查找（Search）**：

```
def search(self, word):
    node = self.root
    for char in word:
        if char not in node.children:
            return False
        node = node.children[char]
    return node.is_end_of_word
```

3. **前缀查询（StartsWith）**：

```
def starts_with(self, prefix):
    node = self.root
    for char in prefix:
        if char not in node.children:
            return False
        node = node.children[char]
    return True
```

四、图

图的表示

```
class Vertex:
    def __init__(self,key):
        self.id=key
        self.connectedTo={}
    def addNeighbor(self,nbr,weight=0):
        self.connectedTo[nbr]=weight
```

```
    def __str__(self):
        return str(self.id)+'connectedTo:'+str([x.id for x in self.connectedTo])
    def getConnections(self):
        return self.connectedTo.keys()
    def getId(self):
        return self.id
    def getWeight(self,nbr):
        return self.connectedTo[nbr]
class Graph:
    def __init__(self):
        self.vertList={}
        self.numVertices=0
    def addVertex(self,key):
        self.numVertices+=1
        newVertex=Vertex(key)
        self.vertList[key]=newVertex
        return newVertex
    def getVertex(self,n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None
    def __contains__(self,n):
        return n in self.vertList
    def addEdge(self,f,t,weight=0):
        if f not in self.vertList:
            nv=self.addVertex(f)
        if t not in self.vertList:
            nv=self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t],weight)
    def getVertices(self):
        return self.vertList.keys()
    def __iter__(self):
        return iter(self.vertList.values())
```

最短路径

dijkstra

Dijkstra算法用于解决单源最短路径问题，即从给定源节点到图中所有其他节点的最短路径。算法 的基本思想是通过不断扩展离源节点最近的节点来逐步确定最短路径。

具体步骤如下：

初始化一个距离数组，用于记录源节点到所有其他节点的最短距离。初始时，源节点的距离为0，其他节点的 距离为无穷大。

选择一个未访问的节点中距离最小的节点作为当前节点。

更新当前节点的邻居节点的距离，如果通过当前节点到达邻居节点的路径比已知最短路径更短，则更新最短路 径。

标记当前节点为已访问。

重复上述步骤，直到所有节点都被访问或者所有节点的最短路径都被确定。

```
  def dijkstra(graph, start):
```

```
    pq = []
    start.distance = 0
    heapq.heappush(pq, (0, start))
    visited = set()
    while pq:
        currentDist, currentVert = heapq.heappop(pq)     # 当一个顶点的最短路径确定后（也就是这个顶点
                                                         # 从优先队列中被弹出时），它的最短路径不会再改变。
        if currentVert in visited:#不要忘记这一步！！！
            continue
        visited.add(currentVert)
        for nextVert in currentVert.getConnections():
            newDist = currentDist + currentVert.getWeight(nextVert)
            if newDist < nextVert.distance:
                nextVert.distance = newDist
                nextVert.pred = currentVert
                heapq.heappush(pq, (newDist, nextVert))
```

Floyd-Warshall

求解所有顶点之间的最短路径可以使用Floyd-Warshall算法，它是一种多源最短路径算法。Floyd-Warshall算法可以在有向图或无向图中找到任意两个顶点之间的最短路径。

算法的基本思想是通过一个二维数组来存储任意两个顶点之间的最短距离。初始时，这个数组包含图中各个顶点之间的直接边的权重，对于不直接相连的顶点，权重为无穷大。然后，通过迭代更新这个数组，逐步求得所有顶点之 间的最短路径。

具体步骤如下：

1. 初始化一个二维数组dist，用于存储任意两个顶点之间的最短距离。初始时，dist[i][j]表示顶点i到顶点j 的直接边的权重，如果i和j不直接相连，则权重为无穷大。
2. 对于每个顶点k，在更新dist数组时，考虑顶点k作为中间节点的情况。遍历所有的顶点对(i, j)，如果通过顶 点k可以使得从顶点i到顶点j的路径变短，则更新dist[i][j]为更小的值。 dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
3. 重复进行上述步骤，对于每个顶点作为中间节点，进行迭代更新dist数组。最终，dist数组中存储的就是 所有顶点之间的最短路径。

Floyd-Warshall算法的时间复杂度为O(V^3)，其中V是图中的顶点数。它适用于解决稠密图（边数较多）的最短路 径问题，并且可以处理负权边和负权回路。

```
def floyd_warshall(graph):
    n = len(graph)
    dist = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if i == j:
                dist[i][j] = 0
            elif j in graph[i]:
                dist[i][j] = graph[i][j]
    for k in range(n):
        for i in range(n):
            for j in range(n):
```

```
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
    return dist
```

最小生成树

可以看到，kruskal 算法的时间复杂度主要来源于对边进行排序，因此其时间复杂度是O(ElogE)，其中E为图的边 数。显然 kruskal 适合顶点数较多、边数较少的情况，这和 prim算法恰好相反。于是可以根据题目所给的数据范 围来选择合适的算法，即如果是**稠密图(边多)，则用 prim 算法;如果是稀疏图(边少)，则用 kruskal 算法。**

prim

步骤:

1. **将起点到所有点的距离设置为无穷，起点入堆**
2. **堆顶元素出堆（排序依据是距离起点的距离）**
3. **访问该节点相邻节点，若首次访问，将相邻节点与起点的距离设置为前次距离和当前距离的最小值**
4. **相邻节点前驱设置为当前节点**
5. **当前节点入树**
6. **相邻节点入堆**

```
import heapq
def prim(graph, n):
    visited = [False] * n
    min_heap = [(0, 0)]  # (weight, vertex)
   min_spanning_tree_cost = 0
   while min_heap:
        weight, vertex = heapq.heappop(min_heap)
       if visited[vertex]:
           continue
       visited[vertex] = True
       min_spanning_tree_cost += weight
       for neighbor, neighbor_weight in graph[vertex]:
           if not visited[neighbor]:
               heapq.heappush(min_heap, (neighbor_weight, neighbor))
   return min_spanning_tree_cost if all(visited) else -1
def main():
   n, m = 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))
   min_spanning_tree_cost = prim(graph, n)
   print(min_spanning_tree_cost)
if __name__ == "__main__":
   main()
```

kruskal

Kruskal算法是一种用于解决最小生成树（Minimum Spanning Tree，简称MST）问题的贪心算法。给定一个连通 的带权无向图，Kruskal算法可以找到一个包含所有顶点的最小生成树，即包含所有顶点且边权重之和最小的树。 以下是Kruskal算法的基本步骤：

1. 将图中的所有边按照权重从小到大进行排序。
2. 初始化一个空的边集，用于存储最小生成树的边。
3. 重复以下步骤，直到边集中的边数等于顶点数减一或者所有边都已经考虑完毕： 选择排序后的边集中权重最小的边。 如果选择的边不会导致形成环路（即加入该边后，两个顶点不在同一个连通分量中），则将该边加入最 小生成树的边集中。
4. 返回最小生成树的边集作为结果。

Kruskal算法的核心思想是通过不断选择权重最小的边，并判断是否会形成环路来构建最小生成树。算法开始时， 每个顶点都是一个独立的连通分量，随着边的不断加入，不同的连通分量逐渐合并为一个连通分量，直到最终形成 最小生成树。

实现Kruskal算法时，一种常用的数据结构是并查集（Disjoint Set）。并查集可以高效地判断两个顶点是否在同一 个连通分量中，并将不同的连通分量合并。

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n
    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]
    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if self.rank[px] > self.rank[py]:
            self.parent[py] = px
        else:
            self.parent[px] = py
            if self.rank[px] == self.rank[py]:
                self.rank[py] += 1
def kruskal(n, edges):
    uf = UnionFind(n)
    edges.sort(key=lambda x: x[2])
    res = 0
    for u, v, w in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            res += w
    if len(set(uf.find(i) for i in range(n))) > 1:
        return -1
    return res
n, m = map(int, input().split())
edges = []
for _ in range(m):
    u, v, w = map(int, input().split())
    edges.append((u, v, w))
print(kruskal(n, edges))
```

判断有无环

dfs

27635: 判断无向图是否连通有无回路(同23163)

http://cs101.openjudge.cn/dsapre/27635/

例题：给定一个无向图，判断是否连通，是否有回路。

输入

第一行两个整数n,m，分别表示顶点数和边数。顶点编号从0到n-1。 (1<=n<=110, 1<=m <= 10000)
接下来m行，每行两个整数u和v，表示顶点u和v之间有边。

输出

如果图是连通的，则在第一行输出“connected:yes",否则第一行输出“connected:no"。
如果图中有回路，则在第二行输出“loop:yes ",否则第二行输出“loop:no"。

样例输入

```
3 2
0 1
0 2
```

样例输出

```
connected:yes
loop:no
```

来源

http://dsbpython.openjudge.cn/dspythonbook/P1040/

```
visited=[]
loop=0
def dfs(pre,begin):
    global loop,visited
    for i in road[begin]:
        if i==pre:
            continue
        if visited[i]:
            loop=1
        else:
            visited[i]=1
            dfs(begin,i)

n,m=map(int,input().split())
road={}
for i in range(n):
    road[i]=[]
for i in range(m):
    u,v=map(int,input().split())
    road[u].append(v)
    road[v].append(u)
```

```
visited=[0 for i in range(n)]
connected=1
visited[0]=1
dfs(-1,0)
for i in range(n):
    if visited[i]==0:
        connected=0
        visited[i]=1
        dfs(-1,i)

if connected:
    print("connected:yes")
else:
    print('connected:no')
if loop:
    print("loop:yes")
else:
    print('loop:no')
```

拓扑排序（kahn）

实现：

Kahn算法的基本思想是通过不断地移除图中的入度为0的顶点，并将其添加到拓扑排序的结果中，直到图中所有的顶点都被移除。具体步骤如下：

1. 初始化一个队列，用于存储当前入度为0的顶点。
2. **遍历图中的所有顶点，计算每个顶点的入度，并将入度为0的顶点加入到队列中。**
3. **不断地从队列中弹出顶点，并将其加入到拓扑排序的结果中。同时，遍历该顶点的邻居，并将其入度减1。如果某个邻居的入度减为0，则将其加入到队列中。**
4. 重复步骤3，直到队列为空。

Kahn算法的时间复杂度为O(V + E)，其中V是顶点数，E是边数。它是一种简单而高效的拓扑排序算法，在有向无环图（DAG）中广泛应用。

```
from collections import deque, defaultdict

def topological_sort(graph):
    indegree = defaultdict(int)
    result = []
    queue = deque()

    # 计算每个顶点的入度
    for u in graph:
        for v in graph[u]:
            indegree[v] += 1

    # 将入度为 0 的顶点加入队列
    for u in graph:
        if indegree[u] == 0:
            queue.append(u)
```

```
    # 执行拓扑排序
    while queue:
        u = queue.popleft()
        result.append(u)

        for v in graph[u]:
            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)

    # 检查是否存在环
    if len(result) == len(graph):
        return result
    else:
        return None
```

强联通子图

Kosaraju's算法可以分为以下几个步骤：

1. **第一次DFS**：对图进行一次DFS，并记录每个顶点的完成时间（即DFS从该顶点返回的时间）。
2. **转置图**：将图中所有边的方向反转，得到转置图。
3. **第二次DFS**：根据第一次DFS记录的完成时间的逆序，对转置图进行DFS。每次DFS遍历到的所有顶点构成一个强连通分量。

详细步骤

1. **第一次DFS**：
   - 初始化一个栈用于记录DFS完成时间顺序。
   - 对图中的每个顶点执行DFS，如果顶点尚未被访问过，则从该顶点开始DFS。
   - DFS过程中，当一个顶点的所有邻居都被访问过后，将该顶点压入栈中。
2. **转置图**：
   - 创建一个新的图，边的方向与原图相反。
3. **第二次DFS**：
   - 初始化一个新的访问标记数组。
   - 根据栈中的顺序（即第一步中记录的完成时间的逆序）对转置图进行DFS。
   - 每次从栈中弹出一个顶点，如果该顶点尚未被访问过，则从该顶点开始DFS，每次DFS遍历到的所有顶点构成一个强连通分量。

以下是Kosaraju's算法的Python实现：

```
from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.graph = defaultdict(list)
        self.V = vertices

    def addEdge(self, u, v):
        self.graph[u].append(v)
```

```
    def _dfs(self, v, visited, stack):
        visited[v] = True
        for neighbour in self.graph[v]:
            if not visited[neighbour]:
                self._dfs(neighbour, visited, stack)
        stack.append(v)

    def _transpose(self):
        g = Graph(self.V)
        for i in self.graph:
            for j in self.graph[i]:
                g.addEdge(j, i)
        return g

    def _fillOrder(self, v, visited, stack):
        visited[v] = True
        for neighbour in self.graph[v]:
            if not visited[neighbour]:
                self._fillOrder(neighbour, visited, stack)
        stack.append(v)

    def _dfsUtil(self, v, visited):
        visited[v] = True
        print(v, end=' ')
        for neighbour in self.graph[v]:
            if not visited[neighbour]:
                self._dfsUtil(neighbour, visited)

    def printSCCs(self):
        stack = []
        visited = [False] * self.V

        for i in range(self.V):
            if not visited[i]:
                self._fillOrder(i, visited, stack)

        gr = self._transpose()

        visited = [False] * self.V

        while stack:
            i = stack.pop()
            if not visited[i]:
                gr._dfsUtil(i, visited)
                print("")

# 示例使用
g = Graph(5)
g.addEdge(1, 0)
g.addEdge(0, 2)
g.addEdge(2, 1)
g.addEdge(0, 3)
g.addEdge(3, 4)

print("Strongly Connected Components:")
```

```
g.printSCCs()
```

五、其他

二分查找

```
import bisect
bisect.bisect_left(a,x, lo=0, hi=len(a)) :
查找在有序列表 a 中插入 x 的index。lo 和 hi 用于指定列表的区间，默认是使用整个列表。如果 x 已经存在，在其左边插入。返回值为 index。
bisect.bisect_right(a,x, lo=0, hi=len(a))
bisect.bisect(a, x,lo=0, hi=len(a)) :
这2个函数和 bisect_left 类似，但如果 x 已经存在，在其右边插入。
bisect.insort_left(a,x, lo=0, hi=len(a)) :
在有序列表 a 中插入 x。和 a.insert(bisect.bisect_left(a,x, lo, hi), x) 的效果相同。
bisect.insort_right(a,x, lo=0, hi=len(a))
bisect.insort(a, x,lo=0, hi=len(a)) :
和 insort_left 类似，但如果 x 已经存在，在其右边插入。
python复制代码import bisect
a = [1, 2, 4, 4, 5]
# 查找插入点
print(bisect.bisect_left(a, 4))  # 输出: 2
print(bisect.bisect_right(a, 4)) # 输出: 4
# 插入元素
bisect.insort_left(a, 3)
print(a)  # 输出: [1, 2, 3, 4, 4, 5]
bisect.insort_right(a, 4)
print(a)  # 输出: [1, 2, 3, 4, 4, 4, 5]
```

math库

向上取整：math.ceil()

向下取整：math.floor()

阶乘：math.factoria()

数学常数：math.pi（圆周率），math.e（自然对数的底）

math.sqrt(x), math.pow(x,y), math.exp(x), math.log(真数，底数)（默认为自然对数）

math.sin(),math.cos(),math.tan()

math.asin(),math.acos(),math.atan()