输入

input() 不读换行符
list(map(int,input().strip().split())) #strip()去前后空格

多组输入，读取到文件末尾EOF结束

while True:
try:
a = input()
except EOFError:
break
z = eval(line) #将line转化为表达式类型并运算

输入不确定行数的数据

x = 0
while True:
try:
x += int(input())
except:
break

正则表达式

1.要转义的：.,^,$,*,+,?,\,[],|,{},()

1. \s 来匹配TAB和空格的混合字符
2. 匹配换行符，TAB，空格的混合字符

4.要match 就要像下面，**注意要用groups()**

```
>>> import re
>>> match = re.match(r"Hello(\s*)(.*)World!", "Hello
>>> match.groups()
('\t\t  ', 'Python ')
```

符号含义

```
模式 描述
^ 匹配字符串的开头
$ 匹配字符串的末尾。
. 匹配任意字符，除了换行符，当re.DOTALL标记被指定时，则可以匹配包括换行符的任意
[...] 用来表示一组字符,单独列出：[amk] 匹配'a'，'m'或'k'
[^...] 不在[]中的字符：[^abc] 匹配除了a,b,c之外的字符。
re* 匹配0个或多个的表达式。
re+ 匹配1个或多个的表达式。
re? 匹配0个或1个由前面的正则表达式定义的片段，非贪婪方式
re{ n} 精确匹配 n 个前面表达式。例如， o{2} 不能匹配 "Bob" 中的"o"，但是能
re{ n,} 匹配 n 个前面表达式。例如， o{2,} 不能匹配"Bob"中的"o"，
但能匹配 "fooooood"中的所有 o。"o{1,}" 等价于 "o+"。"o{0,}"
则等价于 "o*"。
re{ n, m} 匹配 n 到 m 次由前面的正则表达式定义的片段，贪婪方式a| b 匹配a或b
(re) 匹配括号内的表达式，也表示一个组
(?imx) 正则表达式包含三种可选标志：i, m, 或 x 。只影响括号中的区域。
(?-imx) 正则表达式关闭 i, m, 或 x 可选标志。只影响括号中的区域。
(?: re) 类似 (...), 但是不表示一个组
(?imx: re) 在括号中使用i, m, 或 x 可选标志
(?-imx: re) 在括号中不使用i, m, 或 x 可选标志
(?#...) 注释.
(?= re) 前向肯定界定符。如果所含正则表达式，以 ... 表示，在当前位置成功匹配时
匹配引擎根本没有提高；模式的剩余部分还要尝试界定符的右边。
(?! re) 前向否定界定符。与肯定界定符相反；当所含表达式不能在字符串当前位置匹配时
(?> re) 匹配的独立模式，省去回溯。
\w 匹配字母数字及下划线
\W 匹配非字母数字及下划线
\s 匹配任意空白字符，等价于 [\t\n\r\f].
\S 匹配任意非空字符
\d 匹配任意数字，等价于 [0-9].
\D 匹配任意非数字
\A 匹配字符串开始
\Z 匹配字符串结束，如果是存在换行，只匹配到换行前的结束字符串。
\z 匹配字符串结束
\G 匹配最后匹配完成的位置。
\b 匹配一个单词边界，也就是指单词和空格间的位置。例如，
'er\b' 可以匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'。
\B 匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配
"never" 中的 'er'。
\n, \t, 等. 匹配一个换行符。匹配一个制表符。等
\1...\9 匹配第n个分组的内容。
\10 匹配第n个分组的内容，如果它经匹配。否则指的是八进制字符码的表达式。
```

常用正则表达式

```
用户名 /^[a-z0-9_-]{3,16}$/
密码  /^[a-z0-9_-]{6,18}$/
十六进制值   /^#?([a-f0-9]{6}|[a-f0-9]{3})$/
电子邮箱    /^([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6})$/
/^[a-z\d]+(\.[a-z\d]+)*@([\da-z](-[\da-z])?)+(\.{1,2}[a-z]+)+$/
URL /^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]*)*\
IP 地址   /((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|
/^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[
HTML 标签 /^<([a-z]+)([^<]+)*(?:>(.*)<\/\1>|\s+\/>)$/
删除代码\\注释    (?<!http:|\S)//.*$
Unicode编码中的汉字范围 /^[\u2E80-\u9FFF]+$/
```

正则表达式的() [] {}有不同的意思。

() 是为了提取匹配的字符串。表达式中有几个()就有几个相应的匹配字符串。

(\s*)表示连续空格的字符串。

[]是定义匹配的字符范围。比如 [a-zA-Z0-9] 表示相应位置的字符要匹配英文字符和数字。[\s*]表示空格或者*号。

{}一般用来表示匹配的长度，比如 \s{3} 表示匹配三个空格，\s{1,3}表示匹配一到三个空格。

(0-9) 匹配 '0-9' 本身。 [0-9] *匹配数字（注意后面有* ，可以为空）[0-9]+ 匹配数字（注意后面有 +，不可以为空）{1-9} 写法错误。

[0-9]{0,9} 表示长度为 0 到 9 的数字字符串。

圆括号()是组，主要应用在限制多选结构的范围/分组/捕获文本/环视/特殊模式处理

示例：

1、(abc|bcd|cde)，表示这一段是abc、bcd、cde三者之一均可，顺序也必须一致

2、(abc)?，表示这一组要么一起出现，要么不出现，出现则按此组内的顺序出现

3、(?:abc)表示找到这样abc这样一组，但不记录，不保存到$变量中，否则可以通过$x取第几个括号所匹配到的项，比如：(aaa)(bbb)(ccc)(?:ddd)(eee)，可以用$1获取(aaa)匹配到的内容，而$3则获取到了(ccc)匹配到的内容，而$4则获取的是由(eee)匹配到的内容，因为前一对括号没有保存变量

4、a(?=bbb) 顺序环视 表示a后面必须紧跟3个连续的b

5、(?i:xxxx) 不区分大小写 (?s:.*) 跨行匹配.可以匹配回车符

方括号是单个匹配，字符集/排除字符集/命名字符集

示例：
1、[0-3]，表示找到这一个位置上的字符只能是0到3这四个数字，与(abc|bcd|cde)的作用比较类似，但圆括号可以匹配多个连续的字符，而一对方括号只能匹配单个字符

2、[^0-3]，表示找到这一个位置上的字符只能是除了0到3之外的所有字符

3、[:digit:] 0-9 [:alnum:] A-Za-z0-9

()和[]有本质的区别

()内的内容表示的是一个子表达式，()本身不匹配任何东西，也不限制匹配任何东西，只是把括号内的内容作为同一个表达式来处理，例如(ab){1,3}，就表示ab一起连续出现最少1次，最多3次。如果没有括号的话，ab{1,3},就表示a，后面紧跟的b出现最少1次，最多3次。另外，括号在匹配模式中也很重要。这个就不延伸了，LZ有兴趣可以自己查查

[]表示匹配的字符在[]中，并且只能出现一次，并且特殊字符写在[]会被当成普通字符来匹配。例如[(a)]，会匹配(、a、)、这三个字符。

字符串匹配和搜索

问题
你想匹配或者搜索特定模式的文本

解决方案
如果你想匹配的是字面字符串，那么你通常只需要调用基本字符串方法就行， 比如 str.find() , str.endswith() , str.startswith() 或者类似的方法：

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'
>>> # Exact match
>>> text == 'yeah'
```

```
False
>>> # Match at start or end
>>> text.startswith('yeah')
True
>>> text.endswith('no')
False
>>> # Search for the location of the first occurrence
>>> text.find('no')
10
>>>
```

对于复杂的匹配需要使用正则表达式和 re 模块。 为了解释正则表达式的基本原理，假设你想匹配数字格式的日期字符串比如 11/27/2012 ，你可以这样做：

```
>>> text1 = '11/27/2012'
>>> text2 = 'Nov 27, 2012'
>>>
>>> import re
>>> # Simple matching: \d+ means match one or more digits
>>> if re.match(r'\d+/\d+/\d+', text1):
... print('yes')
... else:
... print('no')
...
yes
>>> if re.match(r'\d+/\d+/\d+', text2):
... print('yes')
... else:
... print('no')
...
no
>>>
```

如果你想使用同一个模式去做多次匹配，你应该先将模式字符串预编译为模式对象。比如：

```
>>> datepat = re.compile(r'\d+/\d+/\d+')
>>> if datepat.match(text1):
... print('yes')
... else:
... print('no')
...
yes
>>> if datepat.match(text2):
```

```
... print('yes')
... else:
... print('no')
...
no
```

match() 总是从字符串开始去匹配，如果你想查找字符串任意部分的模式出现位置， 使用 findall() 方法去代替。比如：

```
>>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> datepat.findall(text)
['11/27/2012', '3/13/2013']
>>>
```

在定义正则式的时候，通常会利用括号去捕获分组。比如：

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>>
```

捕获分组可以使得后面的处理更加简单，因为可以分别将每个组的内容提取出来。比如：

```
>>> m = datepat.match('11/27/2012')
>>> m
<_sre.SRE_Match object at 0x1005d2750>
>>> # Extract the contents of each group
>>> m.group(0)
'11/27/2012'
>>> m.group(1)
'11'
>>> m.group(2)
'27'
>>> m.group(3)
'2012'
>>> m.groups()
('11', '27', '2012')
>>> month, day, year = m.groups()
>>>
>>> # Find all matches (notice splitting into tuples)
>>> text
'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> datepat.findall(text)
[('11', '27', '2012'), ('3', '13', '2013')]
>>> for month, day, year in datepat.findall(text):
```

```
... print('{}-{}-{}'.format(year, month, day))
...
2012-11-27
2013-3-13
>>>
```

findall() 方法会搜索文本并以列表形式返回所有的匹配。 如果你想以迭代方式返回匹配，可以使用 finditer() 方法来代替，比如：

```
>>> for m in datepat.finditer(text):
... print(m.groups())
...
('11', '27', '2012')
('3', '13', '2013')
>>>
```

讨论
关于正则表达式理论的教程已经超出了本书的范围。 不过，这一节阐述了使用re模块进行匹配和搜索文本的最基本方法。 核心步骤就是先使用 re.compile() 编译正则表达式字符串， 然后使用 match() , findall() 或者 finditer() 等方法。

当写正则式字符串的时候，相对普遍的做法是使用原始字符串比如 r'(\d+)/(\d+)/(\d+)' 。 这种字符串将不去解析反斜杠，这在正则表达式中是很有用的。如果不这样做的话，你必须使用两个反斜杠，类似 '(\d+)/(\d+)/(\d+)' 。

需要注意的是 match() 方法仅仅检查字符串的开始部分。它的匹配结果有可能并不是你期望的那样。比如：

```
>>> m = datepat.match('11/27/2012abcdef')
>>> m
<_sre.SRE_Match object at 0x1005d27e8>
>>> m.group()
'11/27/2012'
>>>
```

如果你想精确匹配，确保你的正则表达式以$结尾，就像这么这样：

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)$')
>>> datepat.match('11/27/2012abcdef')
>>> datepat.match('11/27/2012')
```

```
<_sre.SRE_Match object at 0x1005d2750>
>>>
```

最后，如果你仅仅是做一次简单的文本匹配/搜索操作的话，可以略过编译部分，直接使用 re 模块级别的函数。比如：

```
>>> re.findall(r'(\d+)/(\d+)/(\d+)', text)
[('11', '27', '2012'), ('3', '13', '2013')]
>>>
```

字符串搜索和替换

问题
你想在字符串中搜索和匹配指定的文本模式

解决方案
对于简单的字面模式，直接使用 str.replace() 方法即可，比如：

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'
>>> text.replace('yeah', 'yep')
'yep, but no, but yep, but no, but yep'
>>>
```

对于复杂的模式，请使用 re 模块中的 sub() 函数。 为了说明这个，假设你想将形式为 11/27/2012 的日期字符串改成 2012-11-27 。示例如下：

```
>>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> import re
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>>
```

sub() 函数中的第一个参数是被匹配的模式，第二个参数是替换模式。反斜杠数字比如 \3 指向前面模式的捕获组号。

如果你打算用相同的模式做多次替换，考虑先编译它来提升性能。比如：

```
>>> import re
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>> datepat.sub(r'\3-\1-\2', text)
'Today is 2012-11-27. PyCon starts 2013-3-13.'
```

```
>>>
```

对于更加复杂的替换，可以传递一个替换回调函数来代替，比如：

```
>>> from calendar import month_abbr
>>> def change_date(m):
... mon_name = month_abbr[int(m.group(1))]
... return '{} {} {}'.format(m.group(2), mon_name, m.group(3))
...
>>> datepat.sub(change_date, text)
'Today is 27 Nov 2012. PyCon starts 13 Mar 2013.'
>>>
```

一个替换回调函数的参数是一个 match 对象，也就是 match() 或者 find() 返回的对象。 使用 group() 方法来提取特定的匹配部分。回调函数最后返回替换字符串。

如果除了替换后的结果外，你还想知道有多少替换发生了，可以使用 re.subn() 来代替。比如：

```
>>> newtext, n = datepat.subn(r'\3-\1-\2', text)
>>> newtext
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>> n
2
>>>
```

字符串忽略大小写的搜索替换

问题
你需要以忽略大小写的方式搜索与替换文本字符串
解决方案
为了在文本操作时忽略大小写，你需要在使用 re 模块的时候给这些操作提供 re.IGNORECASE 标志参数。比如：

```
>>> text = 'UPPER PYTHON, lower python, Mixed Python'
>>> re.findall('python', text, flags=re.IGNORECASE)
['PYTHON', 'python', 'Python']
>>> re.sub('python', 'snake', text, flags=re.IGNORECASE)
'UPPER snake, lower snake, Mixed snake'
>>>
```

最后的那个例子揭示了一个小缺陷，替换字符串并不会自动跟被匹配字符串的大小写保持一致。 为了修复这个，你可能需要一个辅助函数，就像下面的这样：

```
def matchcase(word):
    def replace(m):
        text = m.group()
        if text.isupper():
            return word.upper()
        elif text.islower():
            return word.lower()
        elif text[0].isupper():
            return word.capitalize()
        else:
            return word
    return replace
```

下面是使用上述函数的方法：

```
>>> re.sub('python', matchcase('snake'), text, flags=re.IGNORECA
'UPPER SNAKE, lower snake, Mixed Snake'
>>>
```

译者注： matchcase('snake') 返回了一个回调函数(参数必须是 match 对象)，前面一节提到过， sub() 函数除了接受替换字符串外，还能接受一个回调函数

多行匹配模式

问题
你正在试着使用正则表达式去匹配一大块的文本，而你需要跨越多行去匹配。

解决方案
这个问题很典型的出现在当你用点(.)去匹配任意字符的时候，忘记了点(.)不能匹配换行符的事实。 比如，假设你想试着去匹配C语言分割的注释：

```
>>> comment = re.compile(r'/\*(.*?)\*/')
>>> text1 = '/* this is a comment */'
>>> text2 = '''/* this is a
... multiline comment */
... '''
```

```
>>>
>>> comment.findall(text1)
[' this is a comment ']
>>> comment.findall(text2)
[]
>>>
```

为了修正这个问题，你可以修改模式字符串，增加对换行的支持。比如：

```
>>> comment = re.compile(r'/\*((?:.|\n)*?)\*/')
>>> comment.findall(text2)
[' this is a\n multiline comment ']
>>>
```

在这个模式中，(?:.|\n) 指定了一个非捕获组 (也就是它定义了一个仅仅用来做匹配，而不能通过单独捕获或者编号的组)。

讨论
re.compile() 函数接受一个标志参数叫 re.DOTALL ，在这里非常有用。 它可以让正则表达式中的点(.)匹配包括换行符在内的任意字符。比如：

```
>>> comment = re.compile(r'/\*(.*?)\*/', re.DOTALL)
>>> comment.findall(text2)
[' this is a\n multiline comment ']
```

对于简单的情况使用 re.DOTALL 标记参数工作的很好， 但是如果模式非常复杂或者是为了构造字符串令牌而将多个模式合并起来(2.18节有详细描述)， 这时候使用这个标记参数就可能出现一些问题。 如果让你选择的话，最好还是定义自己的正则表达式模式，这样它可以在不需要额外的标记参数下也能工作的很好。

math

函数

```
acos(x) 返回x的反余弦弧度值。
asin(x) 返回x的反正弦弧度值。
atan(x) 返回x的反正切弧度值。
atan2(y, x) 返回给定的 X 及 Y 坐标值的反正切值。
cos(x)  返回x的弧度的余弦值。
hypot(x, y) 返回欧几里德范数 sqrt(x*x + y*y)。
sin(x)  返回的x弧度的正弦值。
tan(x)  返回x弧度的正切值。
degrees(x)  将弧度转换为角度,如degrees(math.pi/2) ,  返回90.0
```

```
radians(x)  将角度转换为弧度
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) 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。
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
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
sqrt(x) 返回数字x的平方根
```

string

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 i
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
```

python字符串格式化符号:

```
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
  %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( + )
<sp>     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'i
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)

%f指定输出小数点位 print('%4.3f' % 21312.12312) 最小总宽度为4, 小数点后3
```

python三引号允许一个字符串跨多行，字符串中可以包含换行符、制表符以及其他特殊字符。

三引号的语法是一对连续的单引号或者双引号（通常都是成对的用）。

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'
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'

方法	描述
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

方法	描述
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.
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，不过是从右边开始查找.
string.rindex(str, beg=0,end=len (string))	类似于 index()，不过是从右边开始.
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

方法	描述
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 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.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

string.format()使用

```
# -*- coding: UTF-8 -*-

print("网站名：{name}, 地址 {url}".format(name="菜鸟教程", url="www.

# 通过字典设置参数
site = {"name": "菜鸟教程", "url": "www.runoob.com"}
print("网站名：{name}, 地址 {url}".format(**site))

# 通过列表索引设置参数
my_list = ['菜鸟教程', 'www.runoob.com']
```

```
print("网站名：{0[0]}, 地址 {0[1]}".format(my_list))  # "0" 是必须的
```

- python函数

hex() 转十六进制

oct() 转八进制

日期和时间

基本的日期与时间转换

问题
你需要执行简单的时间转换，比如天到秒，小时到分钟等的转换。

解决方案
为了执行不同时间单位的转换和计算，请使用 datetime 模块。 比如，为了表示一个时间段，可以创建一个 timedelta 实例，就像下面这样：

```
>>> from datetime import timedelta
>>> a = timedelta(days=2, hours=6)
>>> b = timedelta(hours=4.5)
>>> c = a + b
>>> c.days
2
>>> c.seconds
37800
>>> c.seconds / 3600
10.5
>>> c.total_seconds() / 3600
58.5
>>>
```

如果你想表示指定的日期和时间，先创建一个 datetime 实例然后使用标准的数学运算来操作它们。比如：

```
>>> from datetime import datetime
>>> a = datetime(2012, 9, 23)
>>> print(a + timedelta(days=10))
2012-10-03 00:00:00
>>>
>>> b = datetime(2012, 12, 21)
```

```
>>> d = b - a
>>> d.days
89
>>> now = datetime.today()
>>> print(now)
2012-12-21 14:54:43.094063
>>> print(now + timedelta(minutes=10))
2012-12-21 15:04:43.094063
>>>
```

在计算的时候，需要注意的是 datetime 会自动处理闰年。比如：

```
>>> a = datetime(2012, 3, 1)
>>> b = datetime(2012, 2, 28)
>>> a - b
datetime.timedelta(2)
>>> (a - b).days
2
>>> c = datetime(2013, 3, 1)
>>> d = datetime(2013, 2, 28)
>>> (c - d).days
1
>>>
```

讨论
对大多数基本的日期和时间处理问题， datetime 模块已经足够了。 如果你需要执行更加复杂的日期操作，比如处理时区，模糊时间范围，节假日计算等等， 可以考虑使用 dateutil模块

许多类似的时间计算可以使用 dateutil.relativedelta() 函数代替。 但是，有一点需要注意的就是，它会在处理月份(还有它们的天数差距)的时候填充间隙。看例子最清楚：

```
>>> a = datetime(2012, 9, 23)
>>> a + timedelta(months=1)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'months' is an invalid keyword argument for this func
>>>
>>> from dateutil.relativedelta import relativedelta
>>> a + relativedelta(months=+1)
datetime.datetime(2012, 10, 23, 0, 0)
>>> a + relativedelta(months=+4)
datetime.datetime(2013, 1, 23, 0, 0)
```

```
>>>
>>> # Time between two dates
>>> b = datetime(2012, 12, 21)
>>> d = b - a
>>> d
datetime.timedelta(89)
>>> d = relativedelta(b, a)
>>> d
relativedelta(months=+2, days=+28)
>>> d.months
2
>>> d.days
28
>>>
```

计算最后一个周五的日期

问题

你需要查找星期中某一天最后出现的日期，比如星期五。

解决方案

Python的 datetime 模块中有工具函数和类可以帮助你执行这样的计算。 下面是对类似这样的问题的一个通用解决方案：

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
"""
Topic: 最后的周五
Desc :
"""
from datetime import datetime, timedelta

weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
            'Friday', 'Saturday', 'Sunday']


def get_previous_byday(dayname, start_date=None):
    if start_date is None:
        start_date = datetime.today()
    day_num = start_date.weekday()
    day_num_target = weekdays.index(dayname)
    days_ago = (7 + day_num - day_num_target) % 7
    if days_ago == 0:
        days_ago = 7
```

```
    target_date = start_date - timedelta(days=days_ago)
    return target_date
```

在交互式解释器中使用如下：

```
>>> datetime.today() # For reference
datetime.datetime(2012, 8, 28, 22, 4, 30, 263076)
>>> get_previous_byday('Monday')
datetime.datetime(2012, 8, 27, 22, 3, 57, 29045)
>>> get_previous_byday('Tuesday') # Previous week, not today
datetime.datetime(2012, 8, 21, 22, 4, 12, 629771)
>>> get_previous_byday('Friday')
datetime.datetime(2012, 8, 24, 22, 5, 9, 911393)
>>>
```

可选的 start_date 参数可以由另外一个 datetime 实例来提供。比如：

```
>>> get_previous_byday('Sunday', datetime(2012, 12, 21))
datetime.datetime(2012, 12, 16, 0, 0)
>>>
```

讨论

上面的算法原理是这样的：先将开始日期和目标日期映射到星期数组的位置上(星期一索引为0)，然后通过模运算计算出目标日期要经过多少天才能到达开始日期。然后用开始日期减去那个时间差即得到结果日期。

如果你要像这样执行大量的日期计算的话，你最好安装第三方包 python-dateutil 来代替。比如，下面是是使用 dateutil 模块中的 relativedelta() 函数执行同样的计算：

```
>>> from datetime import datetime
>>> from dateutil.relativedelta import relativedelta
>>> from dateutil.rrule import *
>>> d = datetime.now()
>>> print(d)
2012-12-23 16:31:52.718111

>>> # Next Friday
>>> print(d + relativedelta(weekday=FR))
2012-12-28 16:31:52.718111
>>>

>>> # Last Friday
>>> print(d + relativedelta(weekday=FR(-1)))
```

```
2012-12-21 16:31:52.718111
>>>
```

计算当前月份的日期范围

问题
你的代码需要在当前月份中循环每一天，想找到一个计算这个日期范围的高效方法。

解决方案

在这样的日期上循环并需要事先构造一个包含所有日期的列表。 你可以先计算出开始日期和结束日期， 然后在你步进的时候使用 datetime.timedelta 对象递增这个日期变量即可。

下面是一个接受任意 datetime 对象并返回一个由当前月份开始日和下个月开始日组成的元组对象。

```
from datetime import datetime, date, timedelta
import calendar

def get_month_range(start_date=None):
    if start_date is None:
        start_date = date.today().replace(day=1)
    _, days_in_month = calendar.monthrange(start_date.year, star
    end_date = start_date + timedelta(days=days_in_month)
    return (start_date, end_date)
```

有了这个就可以很容易的在返回的日期范围上面做循环操作了：

```
>>> a_day = timedelta(days=1)
>>> first_day, last_day = get_month_range()
>>> while first_day < last_day:
...     print(first_day)
...     first_day += a_day
...
2012-08-01
2012-08-02
2012-08-03
2012-08-04
2012-08-05
2012-08-06
2012-08-07
```

```
2012-08-08
2012-08-09
```

讨论
上面的代码先计算出一个对应月份第一天的日期。 一个快速的方法就是使用 date 或 datetime 对象的 replace() 方法简单的将 days 属性设置成1即可。 replace() 方法一个好处就是它会创建和你开始传入对象类型相同的对象。 所以，如果输入参数是一个 date 实例，那么结果也是一个 date 实例。 同样的，如果输入是一个 datetime 实例，那么你得到的就是一个 datetime 实例。

然后，使用 calendar.monthrange() 函数来找出该月的总天数。 任何时候只要你想获得日历信息，那么 calendar 模块就非常有用了。 monthrange() 函数会返回包含星期和该月天数的元组。

一旦该月的天数已知了，那么结束日期就可以通过在开始日期上面加上这个天数获得。 有个需要注意的是结束日期并不包含在这个日期范围内(事实上它是下个月的开始日期)。 这个和Python的 slice 与 range 操作行为保持一致，同样也不包含结尾。

为了在日期范围上循环，要使用到标准的数学和比较操作。 比如，可以利用 timedelta 实例来递增日期，小于号<用来检查一个日期是否在结束日期之前。

理想情况下，如果能为日期迭代创建一个同内置的 range() 函数一样的函数就好了。 幸运的是，可以使用一个生成器来很容易的实现这个目标：

```
def date_range(start, stop, step):
    while start < stop:
        yield start
        start += step
```

下面是使用这个生成器的例子：

```
>>> for d in date_range(datetime(2012, 9, 1), datetime(2012,10,1
                        timedelta(hours=6)):
...     print(d)
...
2012-09-01 00:00:00
2012-09-01 06:00:00
2012-09-01 12:00:00
2012-09-01 18:00:00
```

```
2012-09-02 00:00:00
2012-09-02 06:00:00
...
>>>
```

这种实现之所以这么简单，还得归功于Python中的日期和时间能够使用标准的数学和比较操作符来进行运算。

随机

随机选择

问题
你想从一个序列中随机抽取若干元素，或者想生成几个随机数。

解决方案
random 模块有大量的函数用来产生随机数和随机选择元素。 比如，要想从一个序列中随机的抽取一个元素，可以使用 random.choice() ：

```
>>> import random
>>> values = [1, 2, 3, 4, 5, 6]
>>> random.choice(values)
2
```

为了提取出N个不同元素的样本用来做进一步的操作，可以使用

```
 random.sample() ：

>>> random.sample(values, 2)
[6, 2]
```

如果你仅仅只是想打乱序列中元素的顺序，可以使用 random.shuffle() ：

```
>>> random.shuffle(values)
>>> values
[2, 4, 6, 5, 3, 1]
```

生成随机整数，请使用 random.randint() ：

```
>>> random.randint(0,10)
2
```

为了生成0到1范围内均匀分布的浮点数，使用 random.random() ：

```
>>> random.random()
0.9406677561675867
```

如果要获取N位随机位(二进制)的整数，使用 random.getrandbits() ：

```
>>> random.getrandbits(200)
335837000776573622800628485064121869519521710558559406913275
```

讨论
random 模块使用 Mersenne Twister 算法来计算生成随机数。这是一个确定性算法， 但是你可以通过 random.seed() 函数修改初始化种子。比如：

```
random.seed() # Seed based on system time or os.urandom()
random.seed(12345) # Seed based on integer given
random.seed(b'bytedata') # Seed based on byte data
除了上述介绍的功能，random模块还包含基于均匀分布、高斯分布和其他分布的随机数生成
```

在 random 模块中的函数不应该用在和密码学相关的程序中。 如果你确实需要类似的功能，可以使用ssl模块中相应的函数。 比如， ssl.RAND_bytes() 可以用来生成一个安全的随机字节序列。

优先队列 & 排序

字典有序

1.键有序，dict就是键有序

2.值有序，用collections包里的OrderedDict

优先队列

heapq

```
import heapq
#向堆中插入元素，heapq会维护列表heap中的元素保持堆的性质
heapq.heappush(heap, item)
#heapq把列表x转换成堆
heapq.heapify(x)
#从可迭代的迭代器中返回最大的n个数，可以指定比较的key
#从可迭代的迭代器中返回最小的n个数，可以指定比较的key
nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
print(heapq.nlargest(3, nums)) # Prints [42, 37, 23]
print(heapq.nsmallest(3, nums)) # Prints [-4, 1, 2]
portfolio = [
    {'name': 'IBM', 'shares': 100, 'price': 91.1},
    {'name': 'AAPL', 'shares': 50, 'price': 543.22},
    {'name': 'FB', 'shares': 200, 'price': 21.09},
    {'name': 'HPQ', 'shares': 35, 'price': 31.75},
    {'name': 'YHOO', 'shares': 45, 'price': 16.35},
    {'name': 'ACME', 'shares': 75, 'price': 115.65}
]
cheap = heapq.nsmallest(3, portfolio, key=lambda s: s['price'])
expensive = heapq.nlargest(3, portfolio, key=lambda s: s['price'
#从堆中删除元素，返回值是堆中最小或者最大的元素
heapq.heappop(heap)
```

类类型

类类型，使用的是小于号_lt_，当然没有重写但是有其他的比较函数例如：*le*,*gt*,*cmp*,也是会调用的，和小于号等价的都可以调用(测试了gt)，具体的这些操作之间的关系我也没有研究过。如果类里面没有重写_lt_，会调用其他的比较操作符，从源代码可以看出来，如果没有_lt_，那么会调用_ge_函数。所以可以重写上述的那些函数：

```
class Skill(object):
    def __init__(self,priority,description):
        self.priority = priority
        self.description = description
    def __lt__(self,other):#operator <
        return self.priority < other.priority
    def __ge__(self,other):#oprator >=
        return self.priority >= other.priority
```

```
    def __le__(self,other):#oprator <=
        return self.priority <= other.priority
    def __cmp__(self,other):
        #call global(builtin) function cmp for int
        return cmp(self.priority,other.priority)
    def __str__(self):
        return '(' + str(self.priority)+',\'' + self.description

def heapq_class():
    heap  = []
    heapq.heappush(heap,Skill(5,'proficient'))
    heapq.heappush(heap,Skill(10,'expert'))
    heapq.heappush(heap,Skill(1,'novice'))
    while heap:
        print heapq.heappop(heap),
    print
```

查找最大或最小的 N 个元素

问题
怎样从一个集合中获得最大或者最小的 N 个元素列表？

堆数据结构最重要的特征是 heap[0] 永远是最小的元素。并且剩余的元素可以很容易的通过调用 heapq.heappop() 方法得到， 该方法会先将第一个元素弹出来，然后用下一个最小的元素来取代被弹出元素（这种操作时间复杂度仅仅是 O(log N)，N 是堆大小）。 比如，如果想要查找最小的 3 个元素，你可以这样做：

```
>>> heapq.heappop(heap)
-4
>>> heapq.heappop(heap)
1
>>> heapq.heappop(heap)
2
```

当要查找的元素个数相对比较小的时候，函数 nlargest() 和 nsmallest() 是很合适的。 **如果你仅仅想查找唯一的最小或最大（N=1）的元素的话，那么使用 min() 和 max() 函数会更快些。 类似的，如果 N 的大小和集合大小接近的时候，通常先排序这个集合然后再使用切片操作会更快点（ sorted (items)[:N] 或者是 sorted(items)[-N:] ）。 需要在正确场合使用函数 nlargest() 和 nsmallest() 才能发挥它们的优势（如果 N 快接近集合大小了，那么使用排序操作会更好些）。**

排序

sorted

```
#排序分静态、动态，平常序和指定序
#单级和多级都是静态的
#sorted
###单级排序，数据结构可以是[[]]/[()]
a = [[0,'cc',12],[1,'ab',55],[2,'aa',30],[3,'bb',12]]
b = sorted(a, key = lambda x:x[2], reverse = True)
c = sorted(a, key = lambda x:x[2])
print('b: ',b)
print('c: ',c)
###多级排序
d = sorted(a,key = lambda x:(x[2],x[1]))
print('d: ',d)
####当数据结构是类对象时：
class element(object):
    def __init__(self,id=0,name='',age=0):
        self.id = id
        self.name = name
        self.age = age
    def __lt__(self,other):
        if self.age != other.age:
            return self.age < other.age
        return self.id < other.id
    def __str__(self):
        return "id={0},name={1},age={2}".format(self.id,self.nam

L = [element(0,'cc',12),element(1,'ab',55),element(2,'aa',30),el
l1 = sorted(L)
print('l1:')
for i in l1:
    print(i)
###动态根据用户指定的索引优先级序排序
index = [1,2]
key_set = ''
for i in range(len(index)):
    key_set += 'x['+str(index[i])+']'
    if i != len(index)-1:
        key_set += ','
e = sorted(a, key = lambda x:(eval(key_set)))
print('e: ',e)
###根据用户指定的index排序
class element2(object):
    index = [2,1,3,0]
```

```
    def __init__(self,id=0,name='',age=0):
        self.id = id
        self.name = name
        self.age = age
    def __lt__(self,other):
        return element2.index[self.id] < element2.index[other.id

    def __str__(self):
        return "id={0},name={1},age={2}".format(self.id,self.nam

L2 = [element2(0,'cc',12),element2(1,'ab',55),element2(2,'aa',30
l2 = sorted(L2) #为什么用不了sort？
print('l2: ')
for i in l2:
    print(i)
```

查找

二分查找

```
def BinarySearch(array,t):
    low = 0
    height = len(array)-1
    while low < height:
        mid = (low+height)/2
        if array[mid] < t:
            low = mid + 1

        elif array[mid] > t:
            height = mid - 1

        else:
            return array[mid]

    return -1
if __name__ == "__main__":
    print BinarySearch([1,2,3,34,56,57,78,87],57)
```

三分答案

题意：修路和桥从(0,0)到(x,y),n个数表示有第二行开始有n行表示有n条河，xi是河的起始位置，wi是河的宽度，，C1表示修路每米的花费，C2表示修桥每米的花费，问你最后花费的最少金额?

思路:先把和合并成一条河,然后就三分河岸的高度即可(不过不知道为啥,题目中明明说明了输入只有整数,但设成double才AC,可能是精度问题)

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <algorithm>
#include <stack>
#include <queue>
#include <map>
#include <set>
#include <math.h>
#define max(a,b) (a)>(b)?(a):(b)
#define min(a,b) (a)>(b)?(b):(a)
#define eps 1e-10
#define inf 0x3f3f3f3f
using namespace std;
double xx,x,y,c1,c2;
double sum;
double cal(double yo) {
    double p=sqrt(xx*xx+yo*yo),r=sqrt((x-xx)*(x-xx)+(y-yo)*(y-yo
    return p*c1+r*c2;
}
double solve(double MIN,double MAX){
    double Left=MIN,Right=MAX;
    double mid=(Right+Left)/2,mmid=(mid+Right)/2;
    double Cmid=cal(mid),Cmmid=cal(mmid);
    ///double mid,mmid,Cmid,Cmmid;
    while(fabs(Cmid-Cmmid)>=eps){
        if(Cmid >Cmmid)
            Left=mid;
        else
            Right=mmid;
        mid=(Right+Left)/2;
        mmid=(Right+mid)/2;
        Cmid=cal(mid),Cmmid=cal(mmid);
    }
    return min(Cmid,Cmmid);
}
int main(){
    int n;
    int i;
    while(~scanf("%d%lf%lf%lf%lf",&n,&x,&y,&c1,&c2)){
        sum = 0;
        for(i = 0;i<n;i++){
            double tx,ty;
            scanf("%lf%lf",&tx,&ty);
```

```
            sum+=ty;
        }
        xx=x-sum;
        double ans=solve(0,y);
        printf("%.2f\n",ans);
    }
    return 0;}
```

查询树

```
#线段树通用版 点修改，区间查询。递归建树：O（nlogn） 修改：O（logn） 查询：O
class SegmentTree(object):
    def __init__(self, nums, l, r):
        if l == r:
            self.l = self.r = l
            self.v = nums[l]
            self.Lchild = self.Rchild = None
            return
        self.l = l
        self.r = r
        mid = (r + l) // 2
        self.Lchild = SegmentTree(nums,l,mid)
        self.Rchild = SegmentTree(nums,mid+1,r)
        self.v = self.Lchild.v + self.Rchild.v

    def update(self, p, val):
        mid = (self.r+self.l) // 2
        if self.l == self.r and self.l == p:
            self.v = val
            return
        if self.l == self.r:
            return
        if p <= mid:
            self.Lchild.update(p,val)
        else:
            self.Rchild.update(p,val)
        self.v = self.Lchild.v + self.Rchild.v

    def query(self,ql,qr):
        mid = (self.r+self.l) // 2
        if ql <= self.l and qr >= self.r or self.l == self.r:
            return self.v
        if qr <= mid:
            rtn = self.Lchild.query(ql,qr)
        elif ql > mid:
            rtn = self.Rchild.query(ql,qr)
        else:
```

```
            rtn = self.Lchild.query(ql,qr) + self.Rchild.query(q
        return rtn

if __name__ == "__main__":
    a = SegmentTree([1,2,3,4],0,3)
    p1 = a.query(0,3)
    a.update(1,3)
    p2 = a.query(0,3)
    print(p1,p2)

# 点修改，区间和询问 初始化：O（nlogn） 查询O（logn） 修改 O（logn）
def lowbit(x):
        return x&(-x)

class BIT(object):
    def __init__(self,nums):
        self.length = len(nums)
        self.nums = nums
        self.tree = [0 for _ in range(self.length+1)]
        for i in range(len(nums)):
            k = i + 1
            while k <= self.length:
                self.tree[k] += nums[i]
                k += k&(-k)


    def update(self,p,v):
        diff = v - self.nums[p]
        i = p+1
        while i <= self.length:
            self.tree[i] += diff
            i += lowbit(i)

    def query(self,x,y): #左闭右闭
        rtn = 0
        i = x
        while i > 0:
            rtn -= self.tree[i]
            i -= lowbit(i)
        i = y+1
        while i > 0:
            rtn += self.tree[i]
            i -= lowbit(i)
        return rtn

if __name__ == "__main__":
    test = BIT([1,2,3,4])
    print(test.query(0,3))
    test.update(3,6)
```

```
    print(test.query(0,3))

# T315 Count of Smaller Numbers After Self
#Input: [5,2,6,1]   Output: [2,1,1,0]
# Explanation:
# To the right of 5 there are 2 smaller elements (2 and 1).
# To the right of 2 there is only 1 smaller element (1).
# To the right of 6 there is 1 smaller element (1).
# To the right of 1 there is 0 smaller element.
# 归并求逆序对
# 第一种用到bisect就是从后往前插，插的位置就是前面比它小的个数
import bisect
class Solution(object):
    def countSmaller(self, nums):
        ans = []
        tmp = []
        for i in nums[::-1]:
            p = bisect.bisect_left(tmp,i)
            ans.insert(0,p)
            tmp.insert(p,i)
        return ans

# 第二种是归并求逆序对的正确姿势。
class Solution:
    def countSmaller(self, nums):
        def sort(nums):
            half = len(nums) // 2
            if half:
                left,right = sort(nums[:half]),sort(nums[half:])
                for i in range(len(nums))[::-1]:
                    if right and left and left[-1][1] > right[-1
                        # print(left,right)
                        ans[left[-1][0]] += len(right)
                        nums[i] = left.pop()
                    elif right:
                        nums[i] = right.pop()
                    else:
                        nums[i] = left.pop()
            return nums
        ans = [0]*len(nums)
        sort(list(enumerate(nums)))
        return ans

if __name__ == "__main__":
    test = Solution()
    print(test.countSmaller([5,2,6,1]))

#Given an array nums, we call (i, j) an important reverse pair i
#You need to return the number of important reverse pairs in the
```

```
class Solution:
    def reversePairs(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        cur = nums + [2*x for x in nums]
        cur.sort()
        # print(cur)
        rank = {a: i + 1 for i,a in enumerate(cur)}
        N = len(cur)
        bit = [0 for i in range(N+2)]
        ans = 0
        for i in nums[::-1]:
            rtn = 0
            p = rank[i]-1
            while p > 0:
                rtn += bit[p]
                p -= p & (-p)
            ans += rtn
            p = rank[2*i]
            while p < N:
                bit[p] += 1
                p += p & (-p)
        return ans
if __name__ == "__main__":
    test = Solution()
    print(test.reversePairs([2,4,3,5,1]))
```

图论

```
# #考虑去重边
#dijkstra
#前向星
from collections import deque
class edge:
    def __init__(self, to, next, w):
        self.to = to
        self.next = next
        self.w = w
n = 6 #顶点个数

edges = []
head = [-1 for i in range(n+1)]
ecnt = 0
```

```
def addEdge(f, to, w):
    global edges
    global ecnt
    global head
    e = edge(to, head[f], w)
    edges.append(e)
    head[f] = ecnt
    ecnt += 1


#Bellman-ford算法适用于单源最短路径，图中边的权重可为负数即负权边，但不可以出
#可以判负环
#如果约定有向加权图G不存在负权回路，即最短路径一定存在。就可以直接用来求最短路
# 如果没说存不存在负权回路，可以在执行该算法前做一次拓扑排序，以判断是否存在负权
def Bellman_ford(start):
    #这种写法最差是O(nm)，期望是O(Km),mean(K)<=2就是spfa，嗯spfa就是队列
    Q = deque([start])
    inq = [0 for i in range(n+1)]
    inq[start] = 1
    d = [100000 for i in range(n+1)]
    cnt = [0 for i in range(n+1)] #记录各点入队次数
    while len(Q) > 0:
        cur = Q.popleft()
        inq[cur] = 0
        enm = head[cur]
        while enm != -1:
            e = edges[enm]
            if e.w + d[cur] < d[e.to]:
                d[e.to] = e.w + d[cur]
                Q.extend(e.to)
                inq[e.to] = 1
                cnt[e.to] += 1
                if cnt[e.to] > n:
                    return None #存在负环
            enm = e.next
    return d
#Dijstra
import time
import heapq

class Node:
     def __init__(self, name):
        self.name = name
        self.vis = False
        self.adjacenciesList = []
        self.pre = None
        self.dis = 0x3f3f3f3f
```

```
     def __lt__(self, other):
        return self.dis < other.dis

class Edge:

    def __init__(self, weight, startvertex, endvertex):
        self.weight = weight
        self.startvertex = startvertex
        self.endvertex = endvertex

def calculateshortestpath(vertexlist, startvertex):
    q = []
    startvertex.dis = 0
    heapq.heappush(q, startvertex)

    while q:
        tmp = heapq.heappop(q)
        for edge in tmp.adjacenciesList:
            tempdist = edge.startvertex.dis + edge.weight
            if tempdist < edge.endvertex.dis:
                edge.endvertex.dis = tempdist
                edge.endvertex.pre = edge.startvertex
                heapq.heappush(q,edge.endvertex)
def getshortestpath(t):
    print("The value of it's minimum distance is: ",t.dis),
    node = t
    while node:
        print(node.name),
        node = node.pre
    print(" ")
#主函数部分
node=[]
edge=[]
n=int(raw_input())
m=int(raw_input())
for i in range(n):
     node.append(Node(i+1)) #生成节点的名字
for i in range(m):
     a,b,c = map(int, raw_input().split())  #a到b的距离为c
     edge.append(Edge(c,node[a-1],node[b-1]))
     node[a-1].adjacenciesList.append(edge[i])
begin=time.clock()
calculateshortestpath(node,node[0])
for i in range(n):
 getshortestpath(node[i])
end=time.clock()
end-=begin
print(" ")
print("read :%f s" % end)
```

```
#拓扑排序


#并查集
root = [i for i in range(n+1)]
def findRoot(me):
    if root[me] != me:
        root[me] = findRoot(root[me])
        return root[me]
    return me

def union(a, b):
    ra = findRoot(a)
    rb = findRoot(b)
    if ra != rb:
        root[rb] = ra
        return ra
    return ra



global n
n,m = list(map(int,input().split(" ")))
for i in range(m):
      a, b = list(map(int,input().strip().split()))
      union(a,b)
for i in range(n+1):
    findRoot(i)
print(root)
    f, t, w = list(map(int,input().strip().split()))
    addEdge(f, t, w)
print(Dijkstra(1))


##############################3
#网络流
#增广路 edmonds-karp 就是bfs


from collections import deque

class edge:
    def __init__(self, frm, to, next, flow, cap):
        self.frm = frm
        self.to = to
```

```
        self.next = next
        self.flow = flow
        self.cap = cap

n = 6 #顶点个数

edges = []
head = [-1 for i in range(n+1)]
ecnt = 0

def addEdge(frm, to, cap):
    global edges
    global ecnt
    global head
    e = edge(frm, to, head[frm], 0, cap)
    edges.append(e)
    head[frm] = ecnt
    ecnt += 1
    e = edge(to, frm, head[to], 0, 0)
    edges.append(e)
    head[to] = ecnt
    ecnt += 1
#简单 但不怎么用的maxflow
def Maxflow(s, t):
    a = [0 for i in range(n+1)]
    p = [-1 for i in range(n+1)]

    a[s] = 10000000
    while True:
        Q = deque([])
        Q.append(s)
        while len(Q) != 0:
            cur = Q.popleft()
            enum = head[cur]
            while enum != -1:
                e = edges[enum]
                if a[e.to] == 0 and e.cap > e.flow:
                    a[e.to] = min(a[cur], e.cap - e.flow)
                    p[e.to] = enum
                    Q.append(e.to)
                enum = e.next
            if a[t] == 0:
                break
        if a[t]
        enum = p[t]
        cur = t
        while enum != -1:
            edges[enum].flow += a[cur]
            edges[enum^1].flow -= a[cur]
```

```
            cur = edges[enum].from
            enum = p[edges[enum].from]
        flow += a[t]
    return flow
```

测试

看时间

import time
start = time.time()
run_function()
end = time.time()

对拍

```
import random
f = open('test.txt','w+',encoding = 'UTF-8')
n = 1000
m = random.randint(500, 10000)
f.write(str(n) + ' ' + str(m)+'\n')
for i in range(m):
    a = random.randint(1,n)
    b = random.randint(1,n)
    f.write(str(a) + ' ' + str(b)+'\n')
f.close()
```