语法

位运算符：&按位与；|按位或；^按位异或

`bin(x), oct(x), hex(x)`

```
#一键换词
raw = input().lower()
old, new = input().lower().split()
text = re.sub(r"(?<=\b)"+old+r"(?=\b)", new, raw)
print(". ".join(s.capitalize() for s in text.split(". ")))
```

方法	描述
string.capitalize()	把字符串的第一个字符大写
string.upper()	转换 string 中的小写字母为大写
str.rjust(500, "0")	左面用0补齐500位

datetime模块

```
import datetime
```

datetime模块包含：date类；time类；datetime类；timedelta类

date类

```
a = date(year, month, day)   #定义一个日期对象
a > b; a == b    #运算符重载
a - b     #获取两日期相差的时间，timedelta对象
```

classmethod date.**fromisoformat**(*date_string*)

返回一个对应于以任何有效的 ISO 8601 格式给出的 *date_string* 的 `date`：

```
>>> date.fromisoformat('2019-12-04')
datetime.date(2019, 12, 4)
>>> date.fromisoformat('20191204')
datetime.date(2019, 12, 4)
```

datetime类

classmethod datetime.**strptime**(*date_string*, *format*)

返回一个对应于 *date_string*，根据 *format* 进行解析得到的 `datetime` 对象。

```
>>>datetime.strptime('2015-04-07 04:30:03.628556', "%Y-%m-%d %H:%M:%S.%f")
datetime.datetime(2015, 4, 7, 4, 30, 3, 628556)
```

支持`a + timedelta` `a - b` `a > b`等

timedelta类

```
a = timedelta(days, seconds, microseconds, milliseconds, minutes, hours, weeks)
a.days; a.seconds    #获取天数和秒数（<1天），毫秒数（<1秒）等
```

functools模块

```
import functools
```

functools.**cmp_to_key**(*func*)

将(旧式的)比较函数转换为新式的键函数。比较函数是任何接受两个参数，对它们进行比较，并在**结果为小于时返回一个负数，相等时返回零，大于时返回一个正数**的可调用对象。

functools.**partial**(*func*, */*, **args*, ***keywords*)

它基于一个函数创建一个可调用对象，把原函数的某些参数固定，调用时只需要传递未固定的参数即可。

functools.**reduce**(*function*, *iterable*[, *initializer*])

将两个参数的 *function* 从左至右积累地应用到 *iterable* 的条目，以便将该可迭代对象缩减为单一的值。 例如，`reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` 是计算 `((((1+2)+3)+4)+5)` 的值。

itertools模块

无穷迭代器:

迭代器	实参	结果	示例
`count()`	start, [step]	start, start+step, start+2*step, …	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, … plast, p0, p1, …	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, … 重复无限次或n次	`repeat(10, 3) --> 10 10 10`

根据最短输入序列长度停止的迭代器:

迭代器	实参	结果	示例
`accumulate()`	p [,func]	p0, p0+p1, p0+p1+p2, …	`accumulate([1,2,3,4,5]) --> 1 3 6 10 15`

迭代器	实参	结果	示例
`chain()`	p, q, ...	p0, p1, ... plast, q0, q1, ...	`chain('ABC', 'DEF') --> A B C D E F`
`chain.from_iterable()`	iterable -- 可迭代对象	p0, p1, ... plast, q0, q1, ...	`chain.from_iterable(['ABC', 'DEF']) --> A B C D E F`
`islice()`	seq, [start,] stop [, step]	seq[start:stop:step] 中的元素	`islice('ABCDEFG', 2, None) --> C D E F G`
`pairwise()`	iterable -- 可迭代对象	(p[0], p[1]), (p[1], p[2])	`pairwise('ABCDEFG') --> AB BC CD DE EF FG`

排列组合迭代器：

迭代器	实参	结果
`product()`	p, q, ... [repeat=1]	笛卡尔积，相当于嵌套的for循环
`permutations()`	p[, r]	长度r元组，所有可能的排列，无重复元素
`combinations()`	p, r	长度r元组，有序，无重复元素
`combinations_with_replacement()`	p, r	长度r元组，有序，元素可重复

例子	结果
`product('ABCD', repeat=2)`	`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`	`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`	`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`	`AA AB AC AD BB BC BD CC CD DD`

heapq模块

堆（**小顶堆**）是一棵完全二叉树，其中每个节点的值都小于等于其各个子节点的值。这个使用数组的实现，索引从 0 开始，且对所有的 *k* 都有 `heap[k] <= heap[2*k+1]` 和 `heap[k] <= heap[2*k+2]`。比较时不存在的元素被认为是无限大。堆最有趣的特性在于最小的元素总是在根结点：`heap[0]`。

- heapq.**heapify**(*x*)

  将list *x* 转换成堆，原地，$O(n)$。

- heapq.**heappush**(*heap*, *item*)

  将 *item* 的值加入 *heap* 中，保持堆的不变性。

- heapq.**heappop**(*heap*)

  弹出并返回 *heap* 的最小的元素，保持堆的不变性。如果堆为空，抛出 `IndexError` 。使用 `heap[0]` ，可以只访问最小的元素而不弹出它。

- heapq.**heappushpop**(*heap*, *item*)¶

  将 *item* 放入堆中，然后弹出并返回 *heap* 的最小元素。该组合操作比先调用 `heappush()` 再调用 `heappop()` 运行起来更有效率。

- heapq.**heapreplace**(*heap*, *item*)

  弹出并返回 *heap* 中最小的一项，同时推入新的 *item*。 堆的大小不变。 如果堆为空则引发 `IndexError`。这个单步骤操作比 `heappop()` 加 `heappush()` 更高效，并且在使用固定大小的堆时更为适宜。 pop/push 组合总是会从堆中返回一个元素并将其替换为 *item*。返回的值可能会比新加入的值大。如果不希望如此，可改用 `heappushpop()`。它的 push/pop 组合返回两个值中较小的一个，将较大的留在堆中。

- heapq.**merge**(**iterables*, *key=None*, *reverse=False*)

  将多个已排序的输入合并为一个已排序的输出。 返回已排序的可迭代对象，不会一次性地将数据全部放入内存，并假定每个输入流都是已排序的（从小到大）。具有两个可选参数，它们都必须指定为关键字参数。

  `sorted(itertools.chain(*iterables), key=..., reverse=...)`

- heapq.**nlargest**(*n*, *iterable*, *key=None*)

  从 *iterable* 所定义的数据集中返回前 *n* 个最大元素组成的列表。

- heapq.**nsmallest**(*n*, *iterable*, *key=None*)

  从 *iterable* 所定义的数据集中返回前 *n* 个最小元素组成的列表。

后两个函数在 *n* 值较小时性能最好。 对于更大的值，使用 `sorted()` 函数会更有效率。 此外，当 `n==1` 时，使用内置的 `min()` 和 `max()` 函数会更有效率。 如果需要重复使用这些函数，请考虑将可迭代对象转为真正的堆。

实现大顶堆：Storing `(key(x), x)` instead of `x`, and then accessing `value[1]`. (This can break stability, but `heapq` doesn't promise stability anyway.)

bisect模块

本模块中的函数被设计为定位插入点。这些函数只会调用 `__lt__()` 方法并将返回一个数组的值之间的插入点。

注意：在 3.10 版之后才增加了 *key* 形参

- bisect.**bisect_left**(*a*, *x*, *lo=0*, *hi=len(a)*, ***, *key=None*)

  在 *a* 中找到 *x* 合适的插入点以维持有序。参数 *lo* 和 *hi* 可以被用于确定需要考虑的子集。如果 *x* 已经在 *a* 里存在，那么插入点会在已存在元素之前（也就是左边）。如果 *a* 是列表（list）的话，返回值是可以被放在 `list.insert()` 的第一个参数的。

  返回的插入点 *ip* 将数组 *a* 分为两个切片使得对于左侧切片 `all(elem < x for elem in a[lo : ip])` 为真值而对于右侧切片 `all(elem >= x for elem in a[ip : hi])` 为真值。 **（第一个不小于x的元素序号）**

- bisect.**bisect_right**(*a*, *x*, *lo=0*, *hi=len(a)*, ***, *key=None*)

- bisect.**bisect**(*a*, *x*, *lo=0*, *hi=len(a)*, ***, *key=None*)

  类似于 `bisect_left()`，但是返回的插入点是在 *a* 中任何现有条目 *x* 之后（即其右侧）。返回的插入点 *ip* 将数组 *a* 分为两个切片使得对于左侧切片 `all(elem <= x for elem in a[lo : ip])` 为真值而对于右则切片 `all(elem > x for elem in a[ip : hi])` 为真值。 **（第一个大于x的元素序号）**

- bisect.**insort_left**(*a*, *x*, *lo=0*, *hi=len(a)*, ***, *key=None*)

  按照已排序顺序将 *x* 插入到 *a* 中。此函数首先会运行 `bisect_left()` 来定位一个插入点。 然后，它会在 *a* 上运行 `insert()` 方法。由缓慢的 $O(n)$ 插入步骤主导。

- bisect.**insort_right**(*a*, *x*, *lo=0*, *hi=len(a)*, ***, *key=None*)

- bisect.**insort**(*a*, *x*, *lo=0*, *hi=len(a)*, ***, *key=None*)

  类似于 `insort_left()`，但是会把 *x* 插入到 *a* 中任何现有条目 *x* 之后。此函数首先会运行 `bisect_right()` 来定位一个插入点。 然后，它会在 *a* 上运行 `insert()` 方法。由缓慢的 $O(n)$插入步骤主导。

array模块

在数组对象创建时用单个字符的 *类型码* 来指定：

类型码	C 类型	Python 类型	以字节为单位的最小大小	备注
`'b'`	signed char	int	1	
`'B'`	unsigned char	int	1	
`'u'`	wchar_t	Unicode 字符	2	(1)
`'h'`	signed short	int	2	
`'H'`	unsigned short	int	2	
`'i'`	signed int	int	2	
`'I'`	unsigned int	int	2	
`'l'`	signed long	int	4	
`'L'`	unsigned long	int	4	
`'q'`	signed long long	int	8	
`'Q'`	unsigned long long	int	8	
`'f'`	float	float	4	
`'d'`	double	float	8	

- *class* array.**array**(*typecode*[, *initializer*])

  一个由 *typecode* 限制类型的新数组，并通过可选的 *initializer* 进行初始化。*initializer* 必须为一个列表，或迭代元素类型合适的可迭代对象。

标准化输入/输出

```
import sys
input = sys.stdin.readline
print = sys.stdout.write

#something...

print(str(obj) + '\n')
```

f-string格式字符串

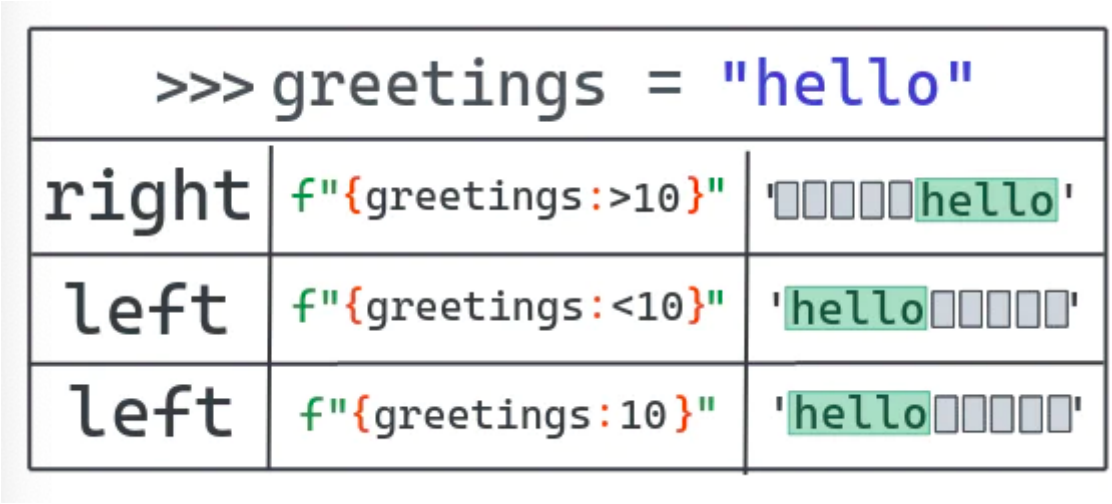
repr的用法

```
name = "Fred"
>>> f"He said his name is {name!r}."
"He said his name is 'Fred'."
>>> f"He said his name is {repr(name)}."  # repr() is equivalent to !r
"He said his name is 'Fred'."
```

十进制数、浮点数的精度表示；嵌套大括号；百分数

```
>>> width = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{width}.{precision}}"  # nested fields
'result:      12.35'
>>> num = 4.123956
>>> f"{num:.2f}"  #浮点数精度
'4.12'
>>>f"{num:+.2f}"
'+4.12'
>>> num = 2343552.6516843
>>> f"{num:,.3f}"
'2,343,552.652'
```

对齐



保持空格

```
>>> foo = "bar"
>>> f"{ foo = }" # preserves whitespace
" foo = 'bar'"
>>> line = "The mill's closed"
>>> f"{line = }"
'line = "The mill\'s closed"'
>>> f"{line = :20}"
"line = The mill's closed   "
>>> f"{line = !r:20}"
'line = "The mill\'s closed" '
```

正则表达式

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字符。
[...]	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"，但是能匹配 "food" 中的两个 o。

模式	描述
re{n,}	匹配 n 个及以上前面表达式。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)	对正则表达式分组并记住匹配的文本
(?= re)	前向肯定界定符。如果所含正则表达式在当前位置成功匹配时成功，否则失败。
(?! 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成功
(?<= re)	后向肯定界定符。
(?<! re)	后向否定界定符。
\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
\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。

- re.**compile**(*pattern*, *flags=0*)

```
prog = re.compile(pattern)
result = prog.match(string)
```

等价于

```
result = re.match(pattern, string)
```

如果需要多次使用这个正则表达式的话，使用 `re.compile()` 保存这个正则对象以便复用，可以让程序更加高效。

- re.**search**(*pattern*, *string*, *flags=0*)

  扫描整个 *string* 查找正则表达式 *pattern* 产生匹配的第一个位置，并返回相应的 `Match`。 如果字符串中没有与模式匹配的位置则返回 `None`。

- re.**match**(*pattern*, *string*, *flags=0*)

  如果 *string* 开头的零个或多个字符与正则表达式 *pattern* 匹配，则返回相应的 `Match`。 如果字符串与模式不匹配则返回 `None`。

- re.**fullmatch**(*pattern*, *string*, *flags=0*)

  如果整个 *string* 与正则表达式 *pattern* 匹配，则返回相应的 `Match`。 如果字符串与模式不匹配则返回 `None`；请注意这与零长度匹配是不同的。*3.4 新版功能.*

- re.**split**(*pattern*, *string*, *maxsplit=0*, *flags=0*)

  用 *pattern* 分开 *string* 。 如果在 *pattern* 中捕获到括号，那么所有的组里的文字也会包含在列表里。

```
re.split(r'\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split(r'\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

- re.**findall**(*pattern*, *string*, *flags=0*)

  返回 *pattern* 在 *string* 中的所有非重叠匹配，以字符串列表或字符串元组列表的形式。

- re.**finditer**(*pattern*, *string*, *flags=0*)

  针对正则表达式 *pattern* 在 *string* 里的所有非重叠匹配返回一个产生 `Match` 对象的 iterator。 *string* 将被从左至右地扫描，并且匹配也将按被找到的顺序返回。 空匹配也会被包括在结果中。*在 3.7 版更改:* 非空匹配现在可以在前一个空匹配之后出现了。

- re.**sub**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

  返回通过使用 *repl* 替换在 *string* 最左边非重叠出现的 *pattern* 而获得的字符串。可选参数 *count* 是要替换的最大次数；*count* 必须是非负整数。如果省略这个参数或设为 0，所有的匹配都会被替换。 样式的空匹配仅在与前一个空匹配不相邻时才会被替换，所以 `sub('x*', '-', 'abxd')` 返回 `'-a-b--d-'`

- re.**subn**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

  行为与 `sub()` 相同，但是返回一个元组 `(字符串, 替换次数)`.*在 3.1 版更改:* 增加了可选标记参数。*在 3.5 版更改:* 不匹配的组合替换为空字符串。

- re.**escape**(*pattern*)

  转义 *pattern* 中的特殊字符。

```
>>>print(re.escape('https://www.python.org'))
https://www\.python\.org
```

Decimal模块

```
from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, ...)

>>> getcontext().prec = 7       # Set a new precision
```

上下文精度和舍入仅在算术运算期间发挥作用。

```
>>> getcontext().prec = 6
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
```

Decimal 数字包括特殊值如代表"非数字"的 `NaN`，正的和负的 `Infinity` 以及 `-0`：

```
>>>Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

- **is_finite**()

  如果参数是一个有限的数，则返回为 `True`；如果参数为无穷大或 NaN，则返回为 `False`。

- **ln**(*context=None*)

  返回操作数的自然对数（以 e 为底）。结果是使用 `ROUND_HALF_EVEN` 舍入模式正确舍入的。

- **log10**(*context=None*)

  返回操作数的以十为底的对数。结果是使用 `ROUND_HALF_EVEN` 舍入模式正确舍入的。

舍入模式：

- decimal.**ROUND_CEILING**

  舍入方向为 `Infinity`。

- decimal.**ROUND_DOWN**

  舍入方向为零。

- decimal.**ROUND_FLOOR**¶

  舍入方向为 `-Infinity`。

- decimal.**ROUND_HALF_EVEN**

  舍入到最接近的数，同样接近则舍入到最接近的偶数。

- decimal.**ROUND_UP**

  舍入到零的反方向。

自定义方法

- object.**init**(*self*[, ...])

  在实例 (通过 `__new__()`) 被创建之后，返回调用者之前调用。其参数与传递给类构造器表达式的参数相同。一个基类如果有 `__init__()` 方法，则其所派生的类如果也有 `__init__()` 方法，就必须显式地调用基类方法以确保实例基类部分的正确初始化；例如: `super().__init__([args...])`

- object.**str**(*self*)

  通过 `str(object)` 以及内置函数 `format()` 和 `print()` 调用以生成一个对象的字符串表示。返回值必须为一个 字符串 对象。

- object.**lt**(*self*, *other*)　object.**le**(*self*, *other*)　object.**eq**(*self*, *other*)　object.**ne**(*self*, *other*)

- object.**gt**(*self*, *other*)　object.**ge**(*self*, *other*)

  以上这些被称为“富比较”方法。运算符号与方法名称的对应关系如下：

  `x<y` 调用 `x.__lt__(y)`、`x<=y` 调用 `x.__le__(y)`、

  `x==y` 调用 `x.__eq__(y)`、`x!=y` 调用 `x.__ne__(y)`、

  `x>y` 调用 `x.__gt__(y)`、`x>=y` 调用 `x.__ge__(y)`。

  如果指定的参数对没有相应的实现，富比较方法可能会返回 `NotImplemented`。实际上这些方法可以返回任意值，因此如果比较运算符是要用于布尔值判断（例如作为 `if` 语句的条件），Python 会对返回值调用 `bool()` 以确定结果为真还是假。

  当**le**、**ge**方法都定义了时，“<=”、“>=”分别调用**le**和**ge**方法，当一个定义另一个未定义时，未定义的操作执行时会调用已经定义的方法求反。**要用sort请定义lt方法**

- object.**len**(*self*)

  调用此方法以实现内置函数 `len()`。 应该返回对象的长度，以一个 `>=` 0 的整数表示。 此外，如果一个对象未定义 `__bool__()` 方法而其 `__len__()` 方法返回值为零则它在布尔运算中将被视为具有假值。

- object.**bool**(*self*)

  调用此方法以实现真值检测以及内置的 `bool()` 操作；应该返回 `False` 或 `True`。 当未定义此方法时，则在定义了 `__len__()` 的情况下将调用它，如果其结果不为零则该对象将被视为具有真值。 如果一个类的 `__len__()` 或 `__bool__()` 均未定义，则其所有实例都将被视为具有真值。

算法

欧拉筛

小于n的素数：

```
n = 10000
prime, cnt, st = [0]*n, 0, [False]*n   #st是素数打标记，primes是额外维护的素数列表
for i in range(2,n):
    if not st[i]:
        prime[cnt] = i
        cnt += 1
    for j in range(n):
        if prime[j]>=n/i: break
        st[prime[j]*i] = True
        if i%prime[j] == 0: break
```

字典树

```
from collections import defaultdict

class Trie:
    def __init__(self):
        self.child = defaultdict(Trie)
        self.exist = None  # 该结点结尾的字符串是否存在

    def insert(self, word):
        curr = self
        for char in word:
            curr = curr.child[char]
        curr.exist = word

    def search(self, string):
        curr = self
        for char in string:
            if char not in curr.child:
                return False
            curr = curr.child[char]
        return curr.exist

    def merge(self, trie2):
        for char, tr in trie2.child.items():
            self.child[char].merge(tr)
        if not self.exist:
            self.exist = trie2.exist

    def is_subseq(self, string):
        for s in string:
            if s in self.child:
                self.merge(self.child.pop(s))
                if barrel.exist:
                    return barrel.exist
        return False
```

Dilworth定理

一个**偏序集**划分成**全序链**的数量，等于最长的**反链**的长度

反链：其中的**任意**两个元素**都不能**比较大小

懒删除堆

```
from heapq import heappush, heappop
from collections import defaultdict

class Barrel(list):
    def __init__(self):
        self.cnt = defaultdict(int)

    def update(self):
        while self and self.cnt[self[0]] == 0:
            heappop(self)

    def push(self, elem):
        if self.cnt[elem] == 0:
            heappush(self, elem)
        self.cnt[elem] += 1

    def delete(self, elem):
        self.cnt[elem] -= 1
```

下面这个更简洁，但需要注意访问堆顶之前需要先调用__bool__

```
class Barrel(list):
    def __init__(self):
        self.cnt = defaultdict(int)

    def __bool__(self):
        while len(self) and self.cnt[self[0]] == 0:
            heappop(self)
        return bool(len(self))

    def push(self, elem):
        if self.cnt[elem] == 0:
            heappush(self, elem)
        self.cnt[elem] += 1

    def delete(self, elem):
        self.cnt[elem] -= 1
```

排列

```
def next_perm(n, num):
    i = n-2
    while num[i] > num[i+1]:
        i -= 1
    j = n-1
    while num[j] < num[i]:
        j -= 1
    num[j], num[i] = num[i], num[j]
    num[i+1:] = num[i+1:][::-1]
```

单调栈

奶牛排队

```
from bisect import bisect_right

h = []
low, high = [], []
ans = 0
for i in range(int(input())):
    h.append(int(input()))
    while low and h[-1] <= h[low[-1]]:
        low.pop()
    low.append(i)
    while high and h[-1] > h[high[-1]]:
        high.pop()
    high.append(i)
    ans = max(ans, i - low[bisect_right(low, high[-2] if len(high)>1 else -1)])
print(ans+1 if ans else 0)
```

BFS

走山路

```
m, n, p = map(int, input().split())
graph = [[int(-1 if s=="#" else s) for s in input().split()] for i in range(m)]
dirc = [[0,1], [1,0], [-1,0], [0,-1]]
for o in range(p):
    sx, sy, ex, ey = map(int, input().split())
    if graph[sx][sy] < 0 or graph[ex][ey] < 0:
        print("NO")
        continue
    record = [[float("inf")]*n for i in range(m)]
    bar = [(0, sx, sy)]
    def bfs():
        while bar:
            t, x, y = heappop(bar)
            if x == ex and y == ey:
                print(t)
                return
```

```
            for i, j in dirc:
                if 0<=x+i<m and 0<=y+j<n and graph[x+i][y+j] != -1:
                    dt = abs(graph[x][y]-graph[x+i][y+j])
                    if record[x+i][y+j] > t+dt:
                        heappush(bar, (t+dt, x+i, y+j))
                        record[x+i][y+j] = t+dt
        print("NO")
    bfs()
```

并查集

```
class Dsu:
    def __init__(self, size):
        self.pa = list(range(size))

    def find(self, x):
        if self.pa[x] != x:
            self.pa[x] = self.find(self.pa[x])
        return self.pa[x]

    def union(self, x, y):
        self.pa[self.find(x)] = self.find(y)
```

调度场

```
import re

pre = {'(':1, '+':2, '-':2, '*':3, '/':3}
for o in range(int(input())):
    s = re.split(r"([^\d.])", input())
    ans = []
    op = []
    for token in s:
        if token == '(':
            op.append('(')
        elif token == ')':
            while (t:=op.pop()) != '(':
                ans.append(t)
        elif token and token in '+-*/':
            while op and pre[op[-1]] >= pre[token]:
                ans.append(op.pop())
            op.append(token)
        elif token:
            ans.append(token)
    while op:
        ans.append(op.pop())
    print(*ans)
```

树的计算

n个结点的二叉树有多少种形态：卡特兰数

$$h(n)=\frac{C_{2n}^{n}}{n+1}$$

n层的AVL树至少有多少结点：$f(x)=f(x-1)+f(x-2)+1$

波兰表达式

波兰表达式是一种把运算符前置的算术表达式，例如普通的表达式2 + 3的波兰表示法为+ 2 3。波兰表达式的优点是运算符之间不必有优先级关系，也不必用括号改变运算次序，例如(2 + 3) * 4的波兰表示法为* + 2 3 4。本题求解波兰表达式的值，其中运算符包括+ - * /四个。

哈夫曼树

```
from heapq import heapify, heappop, heappush

class Node:
    def __init__(self, value, name, child = []):
        self.val = value
        self.name = name
        self.child = child

    def __lt__(self, other):
        return (self.val, self.name) < (other.val, other.name)

    def getCode(self, char, path = []):
        if char == self.name:
            return path
        else:
            for i, nd in enumerate(self.child):
                if p := nd.getCode(char, path + [str(i)]):
                    return p

q = [(lambda char, freq: Node(int(freq), char))(*input().split()) for o in
range(int(input()))]
heapify(q)
while len(q) > 1:
    a, b = heappop(q), heappop(q)
    heappush(q, Node(a.val + b.val, a.name + b.name, child=[a,b]))
while True:
    try:
        s = input()
    except EOFError:
        break
    if s.isdigit():    #解码
        ans = []
        curr = q[0]
        for code in map(int, s + '0'):
            if not curr.child:
                ans.append(curr.name)
                curr = q[0]
```

```
            curr = curr.child[code]
        print(''.join(ans))
    else:     #编码
        print(''.join(code for char in s for code in q[0].getCode(char)))
```