位运算

操作符	名称	说明
`&`	按位与	`1 & 1 = 1`, 其余为 `0`
`|`	按位或	`0 | 0 = 1`, 其余为 1
`^`	按位异或	相同为 `0`, 不同为 `1`
`~`	按位取反	`~x = -x - 1`
`<<`	左移	左移 `n` 位，相当于 `x * (2 ** n)`
`>>`	右移	右移 `n` 位，相当于 `x // (2 ** n)`

可重集 Counter

- 初始化

```python
from collections import Counter

counter = Counter()

# method 1
counter['a'] = 3
counter['b'] = 2
counter['c'] = 1
print(counter) # Counter({'a': 3, 'b': 2, 'c': 1})

# method 2
counter = Counter(['a', 'b', 'c', 'a', 'b', 'a'])
print(counter) # Counter({'a': 3, 'b': 2, 'c': 1})

# method 3
counter = Counter('abcaba')
print(counter) # Counter({'a': 3, 'b': 2, 'c': 1})
```

- 长度

```python
print(len(counter)) # 3
print(sum(counter.values())) # 6
```

- `most_common(n)` : 返回一个列表，包含出现频率最高的 n 个元素和它们的计数.

```
print(counter.most_common(2))  # [('a', 3), ('b', 2)]
```

- `elements()`：返回一个迭代器，其中包含元素，重复的元素会按出现次数重复输出.

```
print(list(counter.elements()))  # ['a', 'a', 'a', 'b', 'b', 'c']
```

- 计数运算

```
counter1 = Counter('aab')
counter2 = Counter('abb')
print(counter1 + counter2) # Counter({'a': 3, 'b': 3})
print(counter1 - counter2) # Counter({'a': 1})
```

- `Counter` 子集判断
-

```
from collections import Counter

c1 = Counter("aabb")     # e.g. {'a': 2, 'b': 2}
c2 = Counter("aaabbbc")  # e.g. {'a': 3, 'b': 3, 'c': 1}

if not (c1 - c2):
    print("c1 is a submultiset of c2")
else:
    print("c1 is not a submultiset of c2")
```

二分法

1. `bisect_left(arr, x, lo=0, hi=len(arr))`:
    - 查找并返回元素 `x` 应该插入到有序列表 `arr` 中的位置.
    - 如果 `x` 已存在于 `arr` 中，它返回 `x` 最左侧的位置（即插入位置）.

```
import bisect
arr = [1, 2, 4, 4, 5]
print(bisect.bisect_left(arr, 4))  # 2

arr = [10, 20, 30, 40, 50]
print(bisect.bisect_left(arr, 35))  # 3
```

2. `bisect_right(arr, x, lo=0, hi=len(arr))`:
    - 与 `bisect_left` 类似，不过如果 `x` 已存在于 `arr` 中，它返回 `x` 最右侧的插入位置.

```
import bisect
arr = [1, 2, 4, 4, 5]
print(bisect.bisect_right(arr, 4))  # 输出: 4
```

3. 参数说明

- `arr`: 一个有序列表.
- `x`: 要查找或插入的元素.
- `lo` 和 `hi`: 可选的范围参数，用来指定查找或插入的子区间，默认为整个列表.

语法糖

`sum` 的用法

1. 将迭代器中元素求和

```
nums = [1, 2, 3, 4, 5]
print(sum(nums)) # output : 15
```

2. 合并 `list`

```
list_of_lists = [[1, 2], [3, 4], [5, 6]]
print(sum(list_of_lists, [])) # output : [1, 2, 3, 4, 5, 6]
```

`zip` 的用法

```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']
res = zip(list1, list2)
print(list(res)) # output : [(1, 'a'), (2, 'b'), (3, 'c')]
```

```
list1 = [1, 2, 3, 4]
list2 = ['a', 'b', 'c']
res = zip(list1, list2)
print(list(res)) # output : [(1, 'a'), (2, 'b'), (3, 'c')]
```

LRU_Cache

```
from functools import lru_cache

@lru_cache(maxsize=None)  # maxsize=None 表示缓存不设上限
def fibonacci(n):
    """
    计算第 n 个斐波那契数（假设 fibonacci(0) = 0, fibonacci(1) = 1）。
    使用 lru_cache 自动缓存，避免重复计算。
    """
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

其中要求参数的可哈希性（hashability）(例如：整数、字符串、元组)