基础语法

字符串

1. 大小写转换

```
text: str
text.upper() # 变全大写
text.lower() # 变全小写
text.capitalize() # 首字母大写
text.title() # 单个字母大写
text.swapcase() # 大小写转换
s[idx].isdigit() # 判断是否为整
s.isnumeric() # 判断是否为数字（包含汉字、阿拉伯数字等）更广泛
```

补充：需要十分注意的一点事，当我们将str转化为list时（如‘sfda’：转化为的是['s', 'f', 'd', 'a']，而不是[‘sfda’])

2. 索引技巧

2.1 列表:

`list.index()`

```
# 返回第一个匹配元素的索引，如果找不到该元素则会引发 ValueError 异常

list.index(element, start, end)

my_list = [10, 20, 30, 40, 50, 30]
index = my_list.index(30)
print(index)  # 输出：2

index = my_list.index(30, 3)
print(index)  # 输出：5
```

```
list(zip(a, b)) # a, b两列表，[1, 2, 4]; [1, 3, 4]=>[[1, 1], [2, 3], [4, 4]]
```

2.2 字符串：

`str.find()` 和 `str.index()`

```
my_string = "Hello, world!"
index = my_string.find("world")
print(index)  # 输出：7

index = my_string.find("Python")
print(index)  # 输出：-1

my_string = "Hello, world!"
index = my_string.index("world")
print(index)  # 输出：7

index = my_string.index("Python")  # 引发 ValueError
```

2.3 字典:

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
exists = 'b' in my_dict
print(exists)  # 输出：True

exists = 'd' in my_dict
print(exists)  # 输出：False

keys_list = list(my_dict.keys())
index = keys_list.index('b')
print(index)  # 输出：1

# 直接查找字典中的键
index = list(my_dict).index('b')
print(index)  # 输出：1

dict.get(key, default=None)
# 返回指定键的值，如果值不在字典中返回default值
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
```

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})
```

bisect

1. **bisect.bisect_left(a, x, lo=0, hi=len(a))**
   - 在列表 `a` 中查找元素 `x` 的插入点，使得插入后仍保持排序。
   - 返回插入点的索引，插入点位于 `a` 中所有等于 `x` 的元素之前。
2. **bisect.bisect_right(a, x, lo=0, hi=len(a))** 或 **bisect.bisect(a, x, lo=0, hi=len(a))**
   - 类似于 `bisect_left`，但插入点位于 `a` 中所有等于 `x` 的元素之后。
3. **bisect.insort_left(a, x, lo=0, hi=len(a))**
   - 在 `a` 中查找 `x` 的插入点并插入 `x`，保持列表 `a` 的有序。
   - 插入点位于 `a` 中所有等于 `x` 的元素之前。
4. **bisect.insort_right(a, x, lo=0, hi=len(a))** 或 **bisect.insort(a, x, lo=0, hi=len(a))**
   - 类似于 `insort_left`，但插入点位于 `a` 中所有等于 `x` 的元素之后。

示例代码

```
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]
```

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))
```

转换

进制

```
b = bin(item)  # 2进制
o = oct(item)  # 8进制
h = hex(item)  # 16进制
```

ASCII

```
ord(char) -> ASCII_value
chr(ascii_value) -> char
```

print保留小数

```
print("%.6f" % x)
print("{:.6f}".format(result))
# 当输出内容很多时：
print('\n'.join(map(str, ans)))
```

算法

1.埃氏筛

```
n = int(input())
prime = [0]*2 + [1]*(n-1)
for i in range(n+1):
    if prime[i]:
```

```
        for j in range(i*i, n+1, i):
            prime[j] = 0
```

2.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。

示例代码

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()
```

代码说明

1. **Graph类**:
   - 初始化图的邻接表表示。
   - `addEdge`方法用于添加图的边。
2. **_dfs和_fillOrder**:

- `_dfs`方法用于深度优先搜索，并在顶点完成时将其添加到栈中。
- `_fillOrder`方法用于填充栈，记录顶点的完成顺序。

3. **_transpose**:
   - `_transpose`方法用于生成转置图。
4. **_dfsUtil**:
   - `_dfsUtil`方法用于在转置图上进行DFS。
5. **printSCCs**:
   - `printSCCs`方法结合上述方法实现Kosaraju's算法，用于打印强连通分量。

在这个实现中，我们首先对原图进行一次DFS，并记录每个顶点的完成时间顺序。然后我们构造转置图，并根据完成时间顺序的逆序对转置图进行DFS，找到所有强连通分量。

二分查找

```
# hi:不可行最小值， lo:可行最大值
lo, hi, ans = 0, max(lst), 0
while lo + 1 < hi:
    mid = (lo + hi) // 2
    # print(lo, hi, mid)
    if check(mid): # 返回True，是因为num>m，是确定不合适
        ans = mid
        lo = mid # 所以lo可以置为 mid + 1。
    else:
        hi = mid
#print(lo)
print(ans)
```

数据结构

单调栈

- 栈中元素单调。若题目时间复杂度降不下来，可以考虑。（Tough预定知识点）

并查集

```
P = list(range(N))
def p(x):
    if P[x] == x:
        return x
    else:
        P[x] = p(P[x])
        return P[x]

def union(x, y):
    px, py = p(x), p(y)
    if px==py:
        return True
    else:
        if <不合法>:  # 根据题意，有时可略
            return False
        else:
```

```
            P[px] = py
            return True
```

8.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
```