目录

Python内置接口

基础数据结构和算法

排序

线性表及其算法：链表、栈、队列；调度场、表达式之间的转

换、单调栈 树：并查集、AVL、Huffman编码树、堆、字典树

图：最短路、最小生成树、拓扑排序

Python内置接口

collections

deque

`deque`（双端队列）是一个从头部和尾部都能快速增删元素的容器。这种数据结构非常适合用于需要快速添加和弹出元素的场景，如队列和栈。

1. 添加元素

- **`append(x)`**：在右端添加一个元素 `x`。时间复杂度为 O(1)。
- **`appendleft(x)`**：在左端添加一个元素 `x`。时间复杂度为 O(1)。

2. 移除元素

- **`pop()`**：移除并返回右端的元素。如果没有元素，将引发 `IndexError`。时间复杂度为 O(1)。
- **`popleft()`**：移除并返回左端的元素。如果没有元素，将引发 `IndexError`。时间复杂度为 O(1)。

3. 扩展

- **`extend(iterable)`**：在右端依次添加 `iterable` 中的元素。整体操作的时间复杂度为 O(k)，其中 `k` 是 `iterable` 的长度。
- **`extendleft(iterable)`**：在左端依次添加 `iterable` 中的元素。注意，添加的顺序会是 `iterable` 元素的逆序。整体操作的时间复杂度为 O(k)，其中 `k` 是 `iterable` 的长度。

4. 其他操作

- **`rotate(n=1)`**：向右旋转队列 `n` 步。如果 `n` 是负数，则向左旋转。这个操作的时间复杂度为 O(k)，其中 `k` 是 `n` 的绝对值，但实际上因为只涉及到指针移动，所以非常快。
- **`clear()`**：移除所有的元素，使其长度为 0。时间复杂度为 O(n)，其中 `n` 是 `deque` 中元素的数量。
- **`remove(value)`**：移除找到的第一个值为 `value` 的元素。这个操作在最坏情况下的时间复杂度为 O(n)，因为可能需要遍历整个 `deque`。

5. 访问元素

- 对于 deque，虽然可以通过索引访问，如 d[0] 或 d[-1]，但这不是 deque 设计的主要用途，且访问中间元素的时间复杂度为 O(n)。因此，如果你需要频繁地从随机位置访问数据，deque 可能不是最佳选择。

```
from collections import deque

# 初始化deque
d = deque([1, 2, 3])

# 添加元素
d.append(4)  # deque变为[1, 2, 3, 4]
d.appendleft(0)  # deque变为[0, 1, 2, 3, 4]

# 移除元素
d.pop()  # 返回 4, deque变为[0, 1, 2, 3]
d.popleft()  # 返回 0, deque变为[1, 2, 3]

# 扩展
d.extend([4, 5])  # deque变为[1, 2, 3, 4, 5]
d.extendleft([0])  # deque变为[0, 1, 2, 3, 4, 5]

# 旋转
d.rotate(1)  # deque变为[5, 0, 1, 2, 3, 4]
d.rotate(-2)  # deque变为[1, 2, 3, 4, 5, 0]

# 清空
d.clear()  # deque变为空
```

Counter, defaultdict, namedtuple, OrderedDict

1. Counter

Counter 是一个用于计数可哈希对象的字典子类。它是一个集合，其中元素的存储形式为字典键值对，键是元素，值是元素计数。

```
from collections import Counter

# 创建 Counter 对象
cnt = Counter(['red', 'blue', 'red', 'green', 'blue', 'blue'])

# 访问计数
print(cnt['blue'])    # 输出: 3
print(cnt['red'])     # 输出: 2

# 更新计数
cnt.update(['blue', 'red', 'blue'])
print(cnt['blue'])    # 输出: 5

# 计数的常见方法
print(cnt.most_common(2))  # 输出 [('blue', 5), ('red', 3)]
```

2. defaultdict

defaultdict 是另一种字典子类，它提供了一个默认值，用于字典所尝试访问的键不存在时返回。

```
from collections import defaultdict

# 使用 lambda 来指定默认值为 0
d = defaultdict(lambda: 0)

d['key1'] = 5
print(d['key1'])  # 输出: 5
print(d['key2'])  # 输出: 0, 因为 key2 不存在, 返回默认值 0
```

3. namedtuple

namedtuple 生成可以使用名字来访问元素内容的元组子类。

```
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])
p = Point(11, y=22)

print(p.x + p.y)  # 输出: 33
print(p[0] + p[1])  # 输出: 33  # 还可以像普通元组那样用索引访问
```

4. OrderedDict

OrderedDict 是一个字典子类，它保持了元素被添加的顺序，这在某些情况下非常有用。

```
from collections import OrderedDict

od = OrderedDict()
od['z'] = 1
od['y'] = 2
od['x'] = 3

for key in od:
    print(key, od[key])
# 输出:
# z 1
# y 2
# x 3
```

permutations

在 Python 中，`permutations` 是 `itertools` 模块中的一个非常有用的函数，用于生成输入可迭代对象的所有可能排列。排列是将一组元素组合成一定顺序的所有可能方式。例如，集合 [1, 2, 3] 的全排列包括 [1, 2, 3]、[1, 3, 2]、[2, 1, 3] 等。

使用 `itertools.permutations`

`itertools.permutations(iterable, r=None)` 函数接收两个参数：

- `iterable`：要排列的数据集。
- `r`：可选参数，指定生成排列的长度。如果 `r` 未指定，则默认值等于 `iterable` 的长度，即生成全排列。

返回值是一个迭代器，生成元组，每个元组是一个可能的排列。

示例代码

下面是使用 `itertools.permutations` 的一些示例：

1. 生成全排列

```
import itertools

data = [1, 2, 3]
permutations_all = list(itertools.permutations(data))

# 输出所有排列
for perm in permutations_all:
    print(perm)
```

输出：

```
(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)
```

2. 生成长度为 `r` 的排列

如果你只想生成一部分元素的排列，可以设置 `r` 的值。

```
import itertools

data = [1, 2, 3, 4]
permutations_r = list(itertools.permutations(data, 2))

# 输出长度为2的排列
```

```
for perm in permutations_r:
    print(perm)
```

输出：

```
(1, 2)
(1, 3)
(1, 4)
(2, 1)
(2, 3)
(2, 4)
(3, 1)
(3, 2)
(3, 4)
(4, 1)
(4, 2)
(4, 3)
```

注意事项

- `itertools.permutations` 生成的排列是 **不重复的**，即使输入的元素中有重复，输出的每个排列仍然是唯一的。
- 生成的排列是按照字典序排列的，基于输入 `iterable` 的顺序。
- 由于排列的数量非常快地随着 `n`（元素总数）和 `r`（排列的长度）的增加而增加，生成非常大的排列集可能会消耗大量的内存和计算资源。例如，10个元素的全排列总共有 10! (即 3,628,800) 种可能，这在实际应用中可能是不切实际的。

使用 `itertools.permutations` 可以有效地处理排列问题，是解决许多算法问题的有力工具。

heapq

`heapq` 模块是 Python 的标准库之一，提供了基于堆的优先队列算法的实现。堆是一种特殊的完全二叉树，满足父节点的值总是小于或等于其子节点的值（在最小堆的情况下）。这个属性使堆成为实现优先队列的理想数据结构。

基本操作

`heapq` 模块提供了一系列函数来管理堆，但它只提供了“最小堆”的实现。以下是一些主要功能及其用法：

1. `heapify(x)`

- **用途**：将列表 `x` 原地转换为堆。
- 示例

```
import heapq
data = [3, 1, 4, 1, 5, 9, 2, 6, 5]
heapq.heapify(data)
print(data)  # 输出将是堆，但可能不是完全排序的
```

2. `heappush(heap, item)`

- **用途**：将 `item` 加入到堆 `heap` 中，并保持堆的不变性。
- 示例

```
heap = []
heapq.heappush(heap, 3)
heapq.heappush(heap, 1)
heapq.heappush(heap, 4)
print(heap)  # 输出最小元素总是在索引0
```

3. `heappop(heap)`

- **用途**：弹出并返回 `heap` 中最小的元素，保持堆的不变性。
- 示例

```
print(heapq.heappop(heap))  # 返回1
print(heap)  # 剩余的堆
```

4. `heapreplace(heap, item)`

- **用途**：弹出堆中最小的元素，并将新的 `item` 插入堆中，效率高于先 `heappop()` 后 `heappush()`。
- 示例

```
heapq.heapreplace(heap, 7)
print(heap)
```

5. `heappushpop(heap, item)`

- **用途**：先将 `item` 压入堆中，然后弹出并返回堆中最小的元素。
- 示例

```
result = heapq.heappushpop(heap, 0)
print(result)  # 输出0
print(heap)  # 剩余的堆
```

6. `nlargest(n, iterable, key=None)` 和 `nsmallest(n, iterable, key=None)`

- **用途**：从 `iterable` 数据中找出最大的或最小的 `n` 个元素。

- 示例

```
data = [3, 1, 4, 1, 5, 9, 2, 6, 5]
print(heapq.nlargest(3, data))  # 输出[9, 6, 5]
print(heapq.nsmallest(3, data))  # 输出[1, 1, 2]
```

应用场景

`heapq` 通常用于需要快速访问最小（或最大）元素的场景，但不需要对整个列表进行完全排序。它广泛应用于数据处理、实时计算、优先级调度等领域。例如，任务调度、Dijkstra 最短路径算法、Huffman 编码树生成等都会用到堆结构。

注意事项

- 如需实现最大堆功能，可以通过对元素取反来实现。将所有元素取负后使用 `heapq`，然后再取负回来即可。
- 堆操作的时间复杂度一般为 O(log n)，适合处理大数据集。
- `heapq` 只能保证列表中的第一个元素是最小的，其他元素的排序并不严格。

queue

Python 的 `queue` 模块提供了多种队列类型，主要用于线程间的通信和数据共享。这些队列都是线程安全的，设计用来在生产者和消费者线程之间进行数据交换。除了已经提到的 `LifoQueue` 之外，`queue` 模块还提供了以下几种有用的队列类型：

1. `Queue`

这是标准的先进先出（FIFO）队列。元素从队列的一端添加，并从另一端被移除。这种类型的队列特别适用于任务调度，保证了任务被处理的顺序。

- **`put(item, block=True, timeout=None)`**：将 `item` 放入队列中。如果可选参数 `block` 设为 `True`，并且 `timeout` 是一个正数，则在超时前会阻塞等待可用的槽位。
- **`get(block=True, timeout=None)`**：从队列中移除并返回一个元素。如果可选参数 `block` 设为 `True`，并且 `timeout` 是一个正数，则在超时前会阻塞等待元素。
- **`empty()`**：判断队列是否为空。
- **`full()`**：判断队列是否已满。
- **`qsize()`**：返回队列中的元素数量。注意，这个大小只是近似值，因为在返回值和队列实际状态间可能存在时间差。

2. `PriorityQueue`

基于优先级的队列，队列中的每个元素都有一个优先级，优先级最低的元素（注意是最“低”）最先被移除。这是通过将元素存储为 `(priority_number, data)` 对来实现的。

- 优先级可以是任何可排序的类型，通常是数字，其中较小的值具有较高的优先级。

3. `SimpleQueue`

在 Python 3.7 及以后版本中引入了 `SimpleQueue`，它是一个简单的先进先出队列，没有大小限制，不像 `Queue`，它没有任务跟踪或其他复杂的功能，通常性能更好。

- `put(item)`：将 `item` 放入队列。
- `get()`：从队列中移除并返回一个元素。
- `empty()`：判断队列是否为空。

4.`LifoQueue`

在 Python 中，LIFO（后进先出）队列可以通过标准库中的 `queue` 模块实现，其中 `LifoQueue` 类提供了一个基于 LIFO 原则的队列实现。LIFO 队列通常被称为堆栈（stack），因为它遵循“后进先出”的原则，即最后一个添加到队列中的元素将是第一个被移除的元素。

`LifoQueue` 提供了以下几个主要的方法：

- `put(item)`: 将 `item` 元素放入队列中。
- `get()`: 从队列中移除并返回最顶端的元素。
- `empty()`: 检查队列是否为空。
- `full()`: 检查队列是否已满。
- `qsize()`: 返回队列中的元素数量。

示例代码

下面是如何使用 `queue.LifoQueue` 的一个简单示例：

```
import queue

# 创建一个 LIFO 队列
lifo_queue = queue.LifoQueue()

# 添加元素
lifo_queue.put('a')
lifo_queue.put('b')
lifo_queue.put('c')

# 依次取出元素
print(lifo_queue.get())  # 输出 'c'
print(lifo_queue.get())  # 输出 'b'
print(lifo_queue.get())  # 输出 'a'
```

注意事项

- `LifoQueue` 是线程安全的，这意味着它可以安全地用于多线程环境。
- 如果 `LifoQueue` 初始化时指定了最大容量，`put()` 方法在队列满时默认会阻塞，直到队列中有空闲位置。如果需要，可以用 `put_nowait()` 方法来避免阻塞，但如果队列满了，这会抛出 `queue.Full` 异常。
- 类似地，`get()` 方法在队列为空时会阻塞，直到队列中有元素可以取出。`get_nowait()` 方法也可以用来避免阻塞，但如果队列空了，会抛出 `queue.Empty` 异常。

示例代码

下面是一个使用 `PriorityQueue` 的例子：

```
import queue

# 创建一个优先级队列
pq = queue.PriorityQueue()

# 添加元素及其优先级
pq.put((3, 'Low priority'))
pq.put((1, 'High priority'))
pq.put((2, 'Medium priority'))

# 依次取出元素
while not pq.empty():
    print(pq.get()[1])  # 输出元素的数据部分
```

使用场景

- **Queue**: 适用于任务调度，如在多线程下载文件时管理下载任务。
- **LifoQueue**: 适用于需要后进先出逻辑的场景，比如回溯算法。
- **PriorityQueue**: 用于需要处理优先级任务的场景，如操作系统的任务调度。
- **SimpleQueue**: 适用于需要快速操作且不需要额外功能的场景，比如简单的数据传递任务。

这些队列因其线程安全的特性，特别适合用于多线程程序中，以确保数据的一致性和完整性。

切片：

```
list[a:b:c]
#a为起始下标，b为结束下标，c为步长，左开右闭

#按照元素的长度进行排序
tele = sorted(tele, key=lambda x: len(x),reverse=True)
```

tips：通过切片可以获得原列表的一个副本，这样后续更改一个列表的内容时不会影响其他副本的值

排序：

```
list=list.sort()#从小到大（按照字典序，如果元素是序列，则从第一个往后依次比较）
list=list.sort(reverse=True)#从大到小
```

插入：

```
list.insert(i,x)#表示在列表的第i索引处插入x，插入后x的索引就是i
```

查找某元素数量：

```

```

```
num1=list.count(a)
```

查找某一元素的下标：

```
str.find(a,b,c)

list.index(a,b,c)
```

其中a表示要找的元素，b表示起始索引，c表示终止索引，左开右闭.

区别：find只适用于字符串，index适用于列表和字符串。

find找不到会返回-1，index找不到会报ValueError

```
#字典
a=dict()
a={}
c=sorted(a)#将字典的键进行排序，返回一个有序的键列表
#a={1:1,1:2}
#c=sorted(a)
#c=[1]

#get() 方法用于获取字典中指定键的值。其语法为 dictionary.get(key, default=None)。如果键存在于字典中，则返回对应的值；如果键不存在，则返回默认值（如果提供了默认值），否则返回 None。这对于避免 KeyError 非常有用
ver.get(vert, None)

#items() 方法用于返回字典的键值对视图。该视图以元组的形式返回字典中的键值对，允许迭代遍历字典中的键值对。语法为 dictionary.items()。例如：
my_dict = {'a': 1, 'b': 2, 'c': 3}
for key, value in my_dict.items():
    print(key, value)
# 输出:
# a 1
# b 2
# c 3
```

去除一个列表中重复的元素：

```
numbers = list(map(int, input().split()))
numbers = list(dict.fromkeys(numbers)) # remove duplicates
```

输出/输入

同一行输出：

使用end=''，引号中间填写希望中间隔开的东西，比如说一个空格

在下一行print()从而达到换行的目的

```
for y in range(m):
    print(long1[x][y],end=" ")
print()

#假设 row 是 [1, 2, 3]，那么 print(*row) 就等价于 print(1, 2, 3)，它会打印出每个元素之间用空格分隔的内容。这种语法对于打印列表、元组等可迭代对象的内容非常方便。

#对于可迭代对象，可以通过*解决
#media=[1,2,3,4]
print(*media)
#输出：1 2 3 4
```

输出列表内所有元素：

```
print(*lst) ##把列表中元素顺序输出
```

保留小数的多种方式：

```
print("{:.2f}".format(3.146)) # 3.15
print(round(3.123456789,5)# 3.12346四舍六入五成双
#保留n位小数：
print(f'{x:.nf}')
```

不定行输入：**套用try-except循环**

Try except 无法使用break来跳出循环，如果已知结束的特定输入（比如说输入0代表结束），

那么可以用以下代码：

```
while True:
    try:
        Xxxxx
        If input()=='0':
        break
    except EOFFError:
        break
```

或者使用sys：

```
import sys
    input = sys.stdin.read().splitlines()
    n, m = map(int, input[0].split())
    prices = list(map(int, input[1].split()))
```

矩阵：

1. 基本方法：列表套列表，用i，j两个指针
2. 将n*n的矩阵用某一元素进行填充的格式：

e.g：a是列数，b是行数

matrix = [[1]*a for _ in range(b)]

3. 双重循环防止index out of range：下界使用max（0，a），上界使用min（n，b），n为矩阵长。注意：range是左开右闭，列表的指针是0-n-1

4. 可以套一层保护圈，如最外圈包一层0，减轻考虑循环的时间

其他

```
float(“inf”)#表示正无穷，可以用于赋值

output = ','.join(my_list)
print(output)#把列表中的所有元素一串输出，条件是列表中的元素都是str类型

mylist=[1,2,3,4,5]
print(mylist,sep=',')#把列表中所有元素之间用sep内部的东西分开，输出形式仍然是一个列表
[1, 2, 3, 4, 5]

ord () #返回对应的ASCII表值
chr () #返回ASCII值对应的字符
bin(),oct(),hex()#分别表示二进制，八进制，十六进制的转换

# 二进制转十进制
binary_str = "1010"
decimal_num = int(binary_str, 2) # 第一个参数是字符串类型的某进制数，第二个参数是他的进制，最终转化为整数
print(decimal_num)  # 输出 10


#Dp写不出来用递归，为了防止爆栈（re）：使用缓存
from functools import lru_cache
@lru_cache(maxsize=128)
#注意：使用的时候函数的参数需要是不可修改的，即不可以为列表字典等，可以改用元组（）或者set

#判断完全平方数
import math
```

```
def isPerfectSquare(num):
    if num < 0:
        return False
    sqrt_num = math.isqrt(num)
    return sqrt_num * sqrt_num == num
print(isPerfectSquare(97)) # False

#年份calendar
import calendar
print(calendar.isleap(2020)) # True，判断是否是闰年

#字符串的连接
str=str1+str2

#用同一个数填充整个列表
dp=[[0]*(i+1) for i in range(5)]
#输出[[0], [0, 0], [0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0, 0]]

#str相关
str.lstrip() / str.rstrip(): 移除字符串左侧/右侧的空白字符。

str.find(sub): 返回子字符串`sub`在字符串中首次出现的索引，如果未找到，则返回-1。

str.replace(old, new): 将字符串中的`old`子字符串替换为`new`。

str.isalpha() / str.isdigit() / str.isalnum(): 检查字符串是否全部由字母/数字/字母和数字组成。

str.title(): 每个单词首字母大写。

#保留小数
#1.round函数
number = 3.14159
rounded_number = round(number, 1)

print(rounded_number)
#输出3.1，保留一位小数
#2.format格式化
number = 3.14159
formatted_number = "{:.1f}".format(number)

print(formatted_number)
#保留一位小数，输出的类型为str

#字符串的连接
''.join()
#eg: ','.join([2,3,4,5])
#output:2,3,4,5

#eval函数
#eval() 是 python 中功能非常强大的一个函数，将字符串当成有效的表达式来求值，并返回计算结果，就是实现 list、dict、tuple、与str 之间的转化
result = eval("1 + 1")
print(result)  # 2
```

```
result = eval("'+' * 5")
print(result)  # +++++

# 3. 将字符串转换成列表
a = "[1, 2, 3, 4]"
result = type(eval(a))
print(result)  # <class 'list'>

input_number = input("请输入一个加减乘除运算公式：")
print(eval(input_number))
## 1*2 +3
## 5



for key,value in dict.items()   #遍历字典的键值对。

for index,value in enumerate(list)  #枚举列表，提供元素及其索引。

dic.setdefault(key,[]).append(value) #常用在字典中加入元素的方式（如果没有值就建空表，有值就直接添加）

dict.get(key,default)   #从字典中获取键对应的值，如果键不存在，则返回默认值`default`。

list(zip(a,b))  #将两个列表元素一一配对，生成元组的列表。
```

类的格式和使用

```
#创建一个类
class person():
    #类变量的创建
    name='aa'
    #类方法的创建
    def who(self):
        print(name)

#类变量的访问：类名.变量名
p=person.name
print(p)


#实例化类（类函数的使用）：类名.函数名
c=person()
c.who()
#输出：aa

#类变量的修改:实例化后只修改自己内部而不影响原始的类；若直接用类名修改则会影响所有的实例化
c=person()
```

```
c.name=0
print(c.name)
#0
a=person()
print(a.name)
#aa
person.name=1
#则后续所有都会改变

#构造器:
class person():
    #self 表示的就是类的实例
    def __init__(self,a)
    #实例变量: 依靠输入
    self.name=a
    #实例变量: 默认值
    self.age=10



print(person(a).name)
#a

#用类实现双端队列:
class deque:
    def __init__(self):
        self.queue=[]

    def push(self,a):#进队
        self.queue.append(a)

    def post_out(self):
        self.queue.pop()

    def pre_out(self):
        self.queue.pop(0)

    def empty(self):
        if self.queue==[]:
            return False
        else:
            return True


t=int(input())
for i in range(t):
    p=deque()
    n=int(input())
    for j in range(n):
            t,x=map(int,input().split())
            if t==1:
                p.push(x)
            elif t==2:
                if x==0:
```

```
                    p.pre_out()
                elif x==1:
                    p.post_out()
    if p.empty():
        ans=[]
        for z in p.queue:
            ans.append(str(z))
        print(' '.join(ans))
    else:
            print('NULL')
```

基础数据结构和算法

1.质数筛

```
#欧拉筛 输出素数列表
def Euler_sieve(n):
    primes = [True for _ in range(n+1)]
    p = 2
    while p*p <= n:
        if primes[p]:
            for i in range(p*p, n+1, p):
                primes[i] = False
        p += 1
    primes[0]=primes[1]=False
    return primes
print(Euler_sieve(20))
# [False, False, True, True, False, True, False, True, False, False,
False, True, False, True, False, False, False, True, False, True, False]

#埃氏筛 输出素数列表
N=20
primes = []
is_prime = [True]*N
is_prime[0] = False;is_prime[1] = False
for i in range(1,N):
    if is_prime[i]:
        primes.append(i)
        for k in range(2*i,N,i): #用素数去筛掉它的倍数
            is_prime[k] = False
print(primes)
# [2, 3, 5, 7, 11, 13, 17, 19]

#欧拉筛 直接判断某个数是否是质数
prime = []
n = c
is_prime=[True] * (n+1)  # 初始化为全是素数
is_prime[0]=is_prime[1]=False#把0和1标记为非素数
```

```
        def euler_sieve():

            for i in range(2, n // 2 + 1):
                if is_prime[i]:
                    prime.append(i)
                for j in prime:  # 将所有质数的倍数标记为非素数

                    if i * j > n:
                        break
                    is_prime[j * i] = False
                    if i % j == 0:
                        break


        # 测试
        euler_sieve()

        if is_prime[n]:
            return True
        else:
            return False

#6k+1质数判断法即Miller-Rabin素性测试算法
import math

def is_prime(num):
    if num <= 1:
        return False
    elif num <= 3:
        return True
    elif num % 2 == 0 or num % 3 == 0:
        return False
    i = 5
    while i * i <= num:
        if num % i == 0 or num % (i + 2) == 0:
            return False
        i += 6
    return True
 #其原理基于以下观察：除了2和3之外，所有的质数都可以表示为 6k ± 1 的形式，其中 k 是一个整数
```

二分查找

```
import bisect
sorted_list = [1,3,5,7,9] #[(0)1, (1)3, (2)5, (3)7, (4)9]
position = bisect.bisect_left(sorted_list, 6)#查找某一元素插入后的索引
print(position)  # 输出：3，因为6应该插入到位置3，才能保持列表的升序顺序

bisect.insort_left(sorted_list, 6)#将某个元素插入原列表并不改变升序/降序
print(sorted_list)  # 输出：[1, 3, 5, 6, 7, 9]，6被插入到适当的位置以保持升序顺序
```

```
sorted_list=(1,3,5,7,7,7,9)
print(bisect.bisect_left(sorted_list,7))
print(bisect.bisect_right(sorted_list,7))
# 输出: 3 6
#右侧插入，如果有相同元素，就输出最大的索引，左侧输入则相反
```

OJ04135:月度开销

http://cs101.openjudge.cn/2024sp_routine/04135/

模拟插板，但是二分 **描述** 农夫约翰是一个精明的会计师。他意识到自己可能没有足够的钱来维持农场的运转了。他计算出并记录下了接下来 N (1 ≤ N ≤ 100,000) 天里每天需要的开销。 约翰打算为连续的M (1 ≤ M ≤ N) 个财政周期创建预算案，他把一个财政周期命名为fajo月。每个fajo月包含一天或连续的多天，每天被恰好包含在一个fajo月里。 约翰的目标是合理安排每个fajo月包含的天数，使得开销最多的fajo月的开销尽可能少。

输入 第一行包含两个整数N,M，用单个空格隔开。 接下来N行，每行包含一个1到10000之间的整数，按顺序给出接下来N天里每天的开销。 输出 一个整数，即最大月度开销的最小值。 样例输入 7 5 100 400 300 100 500 101 400

样例输出 500

```
n, m = map(int, input().split())
L = list(int(input()) for x in range(n))


def check(x):
    num, cut = 1, 0
    for i in range(n):
        if cut + L[i] > x:
            num += 1
            cut = L[i]  # 在L[i]左边插一个板，L[i]属于新的fajo月
        else:
            cut += L[i]
    return num <= m


maxmax = sum(L)
minmax = max(L)
while minmax < maxmax:
    middle = (maxmax + minmax) // 2
    if check(middle):  # 表明这种插法可行，那么看看更小的插法可不可以
        maxmax = middle
    else:
        minmax = middle + 1  # 这种插法不可行，改变minmax看看下一种插法可不可以
print(maxmax)
```

OJ08210:河中跳房子

http://cs101.openjudge.cn/2024sp_routine/08210/

描述 每年奶牛们都要举办各种特殊版本的跳房子比赛，包括在河里从一个岩石跳到另一个岩石。这项激动人心的活动在一条长长的笔直河道中进行，在起点和离起点L远 (1 ≤ L≤ 1,000,000,000) 的终点处均有一个岩石。在起点和终点之间，有N (0 ≤ N ≤ 50,000) 个岩石，每个岩石与起点的距离分别为Di (0 < Di < L)。

在比赛过程中，奶牛轮流从起点出发，尝试到达终点，每一步只能从一个岩石跳到另一个岩石。当然，实力不济的奶牛是没有办法完成目标的。

农夫约翰为他的奶牛们感到自豪并且年年都观看了这项比赛。但随着时间的推移，看着其他农夫的胆小奶牛们在相距很近的岩石之间缓慢前行，他感到非常厌烦。他计划移走一些岩石，使得从起点到终点的过程中，最短的跳跃距离最长。他可以移走除起点和终点外的至多M (0 ≤ M ≤ N) 个岩石。

请帮助约翰确定移走这些岩石后，最长可能的最短跳跃距离是多少?

输入 第一行包含三个整数L, N, M，相邻两个整数之间用单个空格隔开。 接下来N行，每行一个整数，表示每个岩石与起点的距离。岩石按与起点距离从近到远给出，且不会有两个岩石出现在同一个位置。 输出 一个整数，最长可能的最短跳跃距离。 **样例输入** 25 5 2 2 11 14 17 21 **样例输出** 4

排序

归并排序（Merge Sort）

基础知识

时间复杂度:

- **最坏情况**: $O(n\log n)$
- **平均情况**: $O(n\log n)$
- **最优情况**: O*($n$log*n)
- **空间复杂度**: O(n) — 需要额外的内存空间来存储临时数组。
- **稳定性**: 稳定 — 相同元素的相对顺序在排序后不会改变。

应用

- **计算逆序对数**：在一个数组中，如果前面的元素大于后面的元素，则这两个元素构成一个逆序对。归并排序可以在排序过程中修改并计算逆序对的总数。这通过在归并过程中，每当右侧的元素先于左侧的元素被放置到结果数组时，记录左侧数组中剩余元素的数量来实现。
- **排序链表**：归并排序在链表排序中特别有用，因为它可以实现在链表中的有效排序而不需要额外的空间，这是由于链表的节点可以通过改变指针而不是实际移动节点来重新排序。

代码示例

基本代码

```
def merge_sort(unsorted_list):
    if len(unsorted_list) <= 1:
        return unsorted_list
# Find the middle point and devide it
```

```
    middle = len(unsorted_list) // 2
    left_list = unsorted_list[:middle]
    right_list = unsorted_list[middle:]

    left_list = merge_sort(left_list)
    right_list = merge_sort(right_list)
    return list(merge(left_list, right_list))

# Merge the sorted halves
def merge(left_half,right_half):
    res = []
    while len(left_half) != 0 and len(right_half) != 0:
        if left_half[0] < right_half[0]:
            res.append(left_half[0])
            left_half.remove(left_half[0])
        else:
            res.append(right_half[0])
            right_half.remove(right_half[0])
    if len(left_half) == 0:
        res = res + right_half
    else:
        res = res + left_half
    return res
unsorted_list = [64, 34, 25, 12, 22, 11, 90]
print(merge_sort(unsorted_list))
```

对链表进行排序

```
class ListNode:
    def __init__(self, value=0, next=None):
        self.value = value
        self.next = next


def split_list(head):
    if not head or not head.next:
        return head

    # 使用快慢指针找到中点
    slow = head
    fast = head.next  # fast 从 head.next 开始确保分割平均
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # 分割链表为两部分
    mid = slow.next
    slow.next = None

    return head, mid
```

```
def merge_sort(head):
    if not head or not head.next:
        return head

    left, right = split_list(head)
    left = merge_sort(left)
    right = merge_sort(right)
    return merge_lists(left, right)


# 创建链表：4 -> 2 -> 1 -> 3
head = ListNode(4, ListNode(2, ListNode(1, ListNode(3))))

# 排序链表
sorted_head = merge_sort(head)

# 打印排序后的链表
current = sorted_head
while current:
    print(current.value, end=" -> ")
    current = current.next
print("None")
```

###**20018:蚂蚁王国的越野跑** http://cs101.openjudge.cn/2024sp_routine/20018/ **描述** 为了促进蚂蚁家族身体健康，提高蚁族健身意识，蚂蚁王国举行了越野跑。假设越野跑共有N个蚂蚁参加，在一条笔直的道路上进行。N个蚂蚁在起点处站成一列，相邻两个蚂蚁之间保持一定的间距。比赛开始后，N个蚂蚁同时沿着道路向相同的方向跑去。换句话说，这N个蚂蚁可以看作x轴上的N个点，在比赛开始后，它们同时向X轴正方向移动。假设越野跑的距离足够远，这N个蚂蚁的速度有的不相同有的相同且保持匀速运动，那么会有多少对参赛者之间发生“赶超”的事件呢？此题结果比较大，需要定义long long类型。请看备注。

输入 第一行1个整数N。 第2... N +1行：N 个非负整数，按从前到后的顺序给出每个蚂蚁的跑步速度。对于50%的数据，2<=N<=1000。对于100%的数据，2<=N<=100000。 **输出** 一个整数，表示有多少对参赛者之间发生赶超事件。 **样例输入** 5 1 5 10 7 6

5 1 5 5 7 6 **样例输出** 7

8 提示 算需要交换多少次来得到一个排好序的数组，其实就是算逆序对。

```
def main():
    import sys
    input = sys.stdin.read().split()
    ptr = 0
    n = int(input[ptr])
    ptr += 1
    if n == 0:
        return
    speeds = []
    for _ in range(n):
        speeds.append(int(input[ptr]))
        ptr += 1
```

```
    def merge_sort(arr):
        if len(arr) <= 1:
            return arr, 0
        mid = len(arr) // 2
        left, left_cnt = merge_sort(arr[:mid])
        right, right_cnt = merge_sort(arr[mid:])
        merged, merge_cnt = merge(left, right)
        return merged, left_cnt + right_cnt + merge_cnt

    def merge(left, right):
        merged = []
        count = 0
        i = j = 0
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                merged.append(left[i])
                count += len(right) - j
                i += 1
            else:
                merged.append(right[j])
                j += 1
        merged += left[i:]
        merged += right[j:]
        return merged, count

    sorted_arr, total = merge_sort(speeds)
    print(total)

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

快速排序（Quick Sort）

时间复杂度

- **最坏情况**: O*(*n2) — 通常发生在已经排序的数组或基准选择不佳的情况下。
- **平均情况**: O*(*nlog*n)
- **最优情况**: *O*(*nlogn*) — 适当的基准可以保证分割平衡。
- **空间复杂度**: *O*(log*n*) — 主要是递归的栈空间。
- **稳定性**: 不稳定 — 基准点的选择和划分过程可能会改变相同元素的相对顺序。

应用：k-th元素

代码示例

普通快排

```python
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[-1]
        less = [x for x in arr[:-1] if x <= pivot]
        greater = [x for x in arr[:-1] if x > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)

# 示例数组
array = [10, 7, 8, 9, 1, 5]
sorted_array = quicksort(array)
print(sorted_array)
```

在无序列表中选择第k大

```python
def partition(nums, left, right):
    pivot = nums[right]
    i = left
    for j in range(left, right):
        if nums[j] > pivot:  # 注意这里是寻找第k大，所以使用大于号
            nums[i], nums[j] = nums[j], nums[i]
            i += 1
    nums[i], nums[right] = nums[right], nums[i]
    return i

def quickselect(nums, left, right, k):
    if left == right:
        return nums[left]
    pivot_index = partition(nums, left, right)
    if k == pivot_index:
        return nums[k]
    elif k < pivot_index:
        return quickselect(nums, left, pivot_index - 1, k)
    else:
        return quickselect(nums, pivot_index + 1, right, k)

def find_kth_largest(nums, k):
    return quickselect(nums, 0, len(nums) - 1, k - 1)
```

堆排序（Heap Sort）

时间复杂度

- **最坏情况**: O*(*n*log*n)
- **平均情况**: O*(*n*log*n)
- **最优情况**: O*(*n*log*n)

- **空间复杂度**: O(1) — 堆排序是原地排序算法，不需要额外的存储空间。
- **稳定性**: 不稳定 — 堆的维护过程可能会改变相同元素的原始相对顺序。

```
# coding:utf-8
# 交换数据元素
def swap(tree, i, j):
    temp = tree[i]
    tree[i] = tree[j]
    tree[j] = temp


# 调整堆，调整为大根堆
def heapify(tree, i):
    if i >= len(tree):
        return
    c1 = 2 * i + 1
    c2 = 2 * i + 2
    max = i
    if c1 < len(tree) and tree[c1] > tree[max]:
        max = c1
    if c2 < len(tree) and tree[c2] > tree[max]:
        max = c2
    if max != i:
        swap(tree, max, i)
        heapify(tree, max)


# 构造堆
def build_heap(tree):
    last_node = len(tree) - 1
    parent = int((last_node - 1) / 2)
    for i in range(parent, -1, -1):
        heapify(tree, i)


# 堆排序.大根堆，降序
def heap_sort(tree):
    build_heap(tree)
    result = []
    for i in range(len(tree) - 1, -1, -1):
        # 交换根节点与最后一个节点,砍断最后一个节点，重新调整堆
        swap(tree, i, 0)
        result.append(tree.pop())
        heapify(tree, 0)
    return result


# 调整堆，调整为小根堆
def heapify1(tree, i):
    if i >= len(tree):
        return
    c1 = 2 * i + 1
```

```
    c2 = 2 * i + 2
    min = i
    if c1 < len(tree) and tree[c1] < tree[min]:
        min = c1
    if c2 < len(tree) and tree[c2] < tree[min]:
        min = c2
    if min != i:
        swap(tree, min, i)
        heapify(tree, min)


# 构造小根堆
def build_heap1(tree):
    last_node = len(tree) - 1
    parent = int((last_node - 1) / 2)
    for i in range(parent, -1, -1):
        heapify1(tree, i)


# 堆排序.小根堆，升序
def heap_sort1(tree):
    build_heap1(tree)
    result = []
    for i in range(len(tree) - 1, -1, -1):
        # 交换根节点与最后一个节点,砍断最后一个节点，重新调整堆
        swap(tree, i, 0)
        result.append(tree.pop())
        heapify1(tree, 0)
    return result


if __name__ == "__main__":
    tree = [2, 5, 3, 1, 10, 4]
    print("原始数据: ")
    print(tree)
    result = heap_sort(tree)
    print("大根堆排序: ")
    print(result)

    # 小根堆测试用例
    tree = [345, 312, 65, 765, 143, 564]
    print("原始数据: ")
    print(tree)
    result = heap_sort1(tree)
    print("小根堆排序: ")
    print(result)
```

线性表

三种表达式的求值及相互转化

求值：

- 前序表达式（波兰表达式）：栈（最好**从右向左**读，但是反过来也可）
- 后序表达式（逆波兰表达式）：栈（**从左向右**读）
- 中序表达式：Shunting Yard Algorithm

前序表达式和后序表达式向其他任何一种转化：建树

中序表达式->后序表达式：Shunting Yard Algorithm+建树

中序表达式->前序表达式：Shunting Yard Algorithm+建树

OJ24591: 中序表达式转后序表达式

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

描述 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 + / *

Shunting Yard Algorithm的理解：

从左到右遍历中序表达式，

若遇到数字，直接加到输出栈，因为后序遍历中，左右叶节点是最先的

若遇到左括号，加入运算符栈，因为左括号是要建立单独的树，是运算符优先级的一种区分

若遇到右括号，从最后一个开始，将运算符中的东西弹出并加入输出栈，直到遇到左括号，因为这代表这一整个子树的建立。

若遇到运算符，则碰到了非叶节点，弹出栈中任何优先级比当前运算符更高或与当前运算符相等（优先级更高或相等代表子树深度小，所以先输出）的运算符，并将它们添加到输出队列中，然后将自己添加到运算符栈，等待右子树

```
operators = ['+', '-', '*', '/']


def is_num(s):
    for i in operators + ['(', ')']:
```

```
        if i in s:
            return False
    return True


def process(raw_input):
    # convert the raw input into separated sequence
    temp, ans = '', []
    for i in raw_input.strip():
        if is_num(i):
            temp += i
        else:
            if temp:
                ans.append(temp)
            ans.append(i)
            temp = ''
    if temp:
        ans.append(temp)
    return ans


def infix_to_postfix(expression):
    # Shunting Yard Algorithm
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2}
    output_stack, op_stack = [], []
    for i in expression:
        if is_num(i):
            output_stack.append(i)
        elif i == '(':
            op_stack.append(i)
        elif i == ')':
            while op_stack[-1] != '(':
                output_stack.append(op_stack.pop())
            op_stack.pop()
        else:
            while op_stack and op_stack[-1] in operators and precedence[i]
<= precedence[op_stack[-1]]:
                output_stack.append(op_stack.pop())
            op_stack.append(i)
    if op_stack:
        output_stack += op_stack[::-1]
    return output_stack


n = int(input())
for i in range(n):
    tokenized = process(input())
    print(' '.join(infix_to_postfix(tokenized)))
```

单调队列

OJ26978:滑动窗口最大值

http://cs101.openjudge.cn/2024sp_routine/26978/

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        n = len(nums)
        q = collections.deque()
        for i in range(k):
            while q and nums[i] >= nums[q[-1]]:
                q.pop()
            q.append(i)

        ans = [nums[q[0]]]
        for i in range(k, n):
            while q and nums[i] >= nums[q[-1]]:
                q.pop()
            q.append(i)
            while q[0] <= i - k:
                q.popleft()
            ans.append(nums[q[0]])

        return ans
```

单调栈

单调栈（Monotonic Stack）是一种常用的数据结构，通常用于解决一些数组或字符串相关的问题，特别是在需要寻找某个元素左右第一个比它大或小的位置时非常有用。在 Python 中，可以使用列表来实现单调栈的功能

```
def monotonic_stack(nums):
    stack = []  # 用列表模拟栈
    result = [-1] * len(nums)  # 初始化结果列表，默认值为-1

    for i in range(len(nums)):
        # 当栈非空且当前元素比栈顶元素大时，出栈并更新结果
        while stack and nums[i] > nums[stack[-1]]:
            result[stack.pop()] = i

        # 将当前元素的索引入栈
        stack.append(i)

    return result

# 示例
nums = [3, 1, 5, 7, 2, 6]
print("原始数组:", nums)
print("单调递增栈结果:", monotonic_stack(nums))
#这段代码实现了一个单调递增栈，它能够找到数组中每个元素右边第一个比它大的元素的位置。
#输出：
```

```
#原始数组: [3, 1, 5, 7, 2, 6]
#单调递增栈结果: [2, 2, 3, -1, 5, -1]

#要实现单调递减栈，只需在 while 循环中将 nums[i] > nums[stack[-1]] 改为 nums[i] < nums[stack[-1]] 即可。
```

OJ28203:【模板】单调栈

http://cs101.openjudge.cn/practice/28203/

给出项数为 n 的整数数列 a1...an。定义函数 f(i) 代表数列中第 i 个元素之后第一个大于 ai 的元素的**下标**。若不存在，则 f(i)=0。试求出 f(1...n)。

输入:第一行一个正整数 n。第二行 n 个正整数 a1...an。

输出:一行 n 个整数表示 f(1), f(2), ..., f(n) 的值。

```
n = int(input())
a = list(map(int, input().split()))
stack = []
for i in range(n):
    while stack and a[stack[-1]] < a[i]:
        a[stack.pop()] = i + 1
    stack.append(i)
while stack:
    a[stack[-1]] = 0
    stack.pop()
print(*a)
```

OJ04137:最小新整数

描述 给定一个十进制正整数n(0 < n < 1000000000)，每个数位上数字均不为0。n的位数为m。 现在从m位中删除k位(0 < k < m), 使得删除后的数最小

输入 第一行t, 表示有t组数据(t <= 10)； 接下来t行，每一行表示一组测试数据，每组测试数据包含两个数字n, k。 **输出** t行，每行一个数字，表示从n中删除k位后得到的最小整数。 **样例输入** 2 9128456 2 1444 3 **样例输出** 12456 1

http://cs101.openjudge.cn/practice/04137/

```
def removeKDigits(num, k):
    stack = []
    for digit in num:
        while k and stack and stack[-1] > digit:
            stack.pop()
            k -= 1
        stack.append(digit)
    while k:
        stack.pop()
```

```
            k -= 1
        return int(''.join(stack))
    t = int(input())
    results = []
    for _ in range(t):
        n, k = input().split()
        results.append(removeKDigits(n, int(k)))
    for result in results:
        print(result)
```

OJ27205:护林员盖房子 加强版

http://cs101.openjudge.cn/2024sp_routine/27205/

描述 在一片保护林中，护林员想要盖一座房子来居住，但他不能砍伐任何树木。 现在请你帮他计算：保护林中所能用来盖房子的矩形空地的最大面积。

输入 保护林用一个二维矩阵来表示，长宽都不超过1000（即<=1000）。 第一行是两个正整数m,n，表示矩阵有m行n列。 然后是m行，每行n个整数，用1代表树木，用0表示空地。 **输出** 一个正整数，表示保护林中能用来盖房子的最大矩形空地面积。 **样例输入** 4 5 0 1 0 1 1 0 1 0 0 1 0 0 0 0 0 1 1 0 1 **样例输出** 5 题解：https://zhuanlan.zhihu.com/p/162834671

```
def maximalRectangle(matrix) -> int:
    if (rows := len(matrix)) == 0:
        return 0

    cols = len(matrix[0])
    # 存储每一层的高度
    height = [0 for _ in range(cols + 1)]
    res = 0

    for i in range(rows):  # 遍历以哪一层作为底层
        stack = [-1]
        for j in range(cols + 1):
            # 计算j位置的高度，如果遇到1则置为0，否则递增
            h = 0 if j == cols or matrix[i][j] == '1' else height[j] + 1
            height[j] = h
            # 单调栈维护长度
            while len(stack) > 1 and h < height[stack[-1]]:
                res = max(res, (j - stack[-2] - 1) * height[stack[-1]])
                stack.pop()
            stack.append(j)
    return res


rows, _ = map(int, input().split())
a = [input().split() for _ in range(rows)]

print(maximalRectangle(a))
```

散列表

OJ17968:整型关键字的散列映射

http://cs101.openjudge.cn/2024sp_routine/17968/ **描述** 给定一系列整型关键字和素数P，用除留余数法定义的散列函数H（key)=key%M，将关键字映射到长度为M的散列表中，用线性探查法解决冲突

输入 输入第一行首先给出两个正整数N（N<=1000）和M（>=N的最小素数），分别为待插入的关键字总数以及散列表的长度。 第二行给出N个整型的关键字。数字之间以空格分隔。 **输出** 在一行内输出每个整型关键字的在散列表中的位置。数字间以空格分隔。 **样例输入** 4 5 24 13 66 77 **样例输出** 4 3 1 2

```
import sys
input = sys.stdin.read

data = input().split()
index = 0
N = int(data[index])
index += 1
M = int(data[index])
index += 1


k = [0.5] * M
l = list(map(int, data[index:index + N]))

ans = []
for u in l:
    t = u % M
    i = t
    while True:
        if k[i] == 0.5 or k[i] == u:
            ans.append(i)
            k[i] = u
            break
        i = (i + 1) % M

print(*ans)
```

OJ17975:用二次探查法建立散列表

http://cs101.openjudge.cn/2024sp_routine/17975/ **描述** 给定一系列整型关键字和素数P，用除留余数法定义的散列函数H（key)=key%M，将关键字映射到长度为M的散列表中，用二次探查法解决冲突.

本题不涉及删除，且保证表长不小于关键字总数的2倍，即没有插入失败的可能。 **输入** 输入第一行首先给出两个正整数N（N<=1000）和M（一般为>=2N的最小素数），分别为待插入的关键字总数以及散列表的长度。 第二行给出N个整型的关键字。数字之间以空格分隔。 **输出** 在一行内输出每个整型关键字的在散列表中的位置。数字间以空格分隔。 **样例输入** 5 11 24 13 35 15 14 **样例输出** 2 3 1 4 7 **提示** 探查增量序列依次为：1^2，-1^2，2^2，-2^2，....,^2表示平方

```
import sys
input = sys.stdin.read
data = input().split()
index = 0
n = int(data[index])
index += 1
m = int(data[index])
index += 1
num_list = [int(i) for i in data[index:index+n]]
mylist = [0.5] * m
def generate_result():
    for num in num_list:
        pos = num % m
        current = mylist[pos]
        if current == 0.5 or current == num:
            mylist[pos] = num
            yield pos
        else:
            sign = 1
            cnt = 1
            while True:
                now = pos + sign * (cnt ** 2)
                current = mylist[now % m]
                if current == 0.5 or current == num:
                    mylist[now % m] = num
                    yield now % m
                    break
                sign *= -1
                if sign == 1:
                    cnt += 1
result = generate_result()
print(*result)
```

树

树的四种遍历及其相互转化

给出二叉树的前序、中序、后序遍历中的两种，建树或者求出另外一种，算法实现麻烦但是思路简单。

树的bfs遍历（或者叫层次遍历，但是我不喜欢这个叫法）和队列有奇妙的联系，见例题

OJ22158:根据二叉树前中序序列建树

http://cs101.openjudge.cn/2024sp_routine/22158/ **描述** 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。 给定二叉树的前序遍历序列和中序遍历序列(长度均不超过26)，请输出该二叉树的后序遍历序列

输入 多组数据 每组数据2行，第一行是前序遍历序列，第二行是中序遍历序列 **输出** 对每组序列建树，输出该树的后序遍历序列 **样例输入** DURPA RUDPA XTCNB CTBNX **样例输出** RUAPD CBNTX

```
class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

    def post_order(self):
        post = ''
        if self.left:
            post += self.left.post_order()
        if self.right:
            post += self.right.post_order()
        return post + self.val


def build(pre_order, in_order):
    if not pre_order or not in_order:
        return None
    #print(pre_order, in_order)
    if len(pre_order) == 1:
        return Node(pre_order[0])
    root_val = pre_order[0]
    div = 0
    while in_order[div] != root_val:
        div += 1
    left_in_order = in_order[: div]
    right_in_order = in_order[div+1:]
    div = 1
    while pre_order[div] in left_in_order:
        div += 1
        if div >= len(pre_order):
            div = len(pre_order)
            break
    return Node(root_val, left=build(pre_order[1: div], left_in_order),
right=build(pre_order[div:], right_in_order))


while True:
    try:
        p, i = input(), input()
        tree = build(p, i)
        print(tree.post_order())
    except EOFError:
        break
```

OJ24750:根据二叉树中后序序列建树

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

```
class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

    def post_order(self):
        post = ''
        if self.left:
            post += self.left.post_order()
        if self.right:
            post += self.right.post_order()
        return post + self.val


def build(pre_order, in_order):
    if not pre_order or not in_order:
        return None
    #print(pre_order, in_order)
    if len(pre_order) == 1:
        return Node(pre_order[0])
    root_val = pre_order[0]
    div = 0
    while in_order[div] != root_val:
        div += 1
    left_in_order = in_order[: div]
    right_in_order = in_order[div+1:]
    div = 1
    while pre_order[div] in left_in_order:
        div += 1
        if div >= len(pre_order):
            div = len(pre_order)
            break
    return Node(root_val, left=build(pre_order[1: div], left_in_order),
right=build(pre_order[div:], right_in_order))


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()
```

OJ25140:根据后序表达式建立表达式树

http://cs101.openjudge.cn/2024sp_routine/25140/ **描述** 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

```
class Node:
    def __init__(self, name, left=None, right=None):
        self.name = name
        self.left = left
        self.right = right


def build(s):
    stack = []
    for i in s:
        if ord(i) > ord('Z'):
            stack.append(Node(i))
        else:
            r, l = stack.pop(), stack.pop()
            stack.append(Node(i, l, r))
    return stack[0]
```

```
for _ in range(int(input())):
    s = input()
    tree = build(s)
    bfs = [tree]
    ans = ''
    while bfs:
        now = bfs.pop(0)
        ans += now.name
        if now.left:
            bfs.append(now.left)
        if now.right:
            bfs.append(now.right)
    print(ans[::-1])
```

并查集

并查集（Union-Find 或 Disjoint Set Union，简称DSU）是一种处理不交集合的合并及查询问题的数据结构。它支持两种操作：

1. **Find**: 确定某个元素属于哪一个子集。这个操作可以用来判断两个元素是否属于同一个子集。
2. **Union**: 将两个子集合并成一个集合。

使用场景

并查集常用于处理一些元素分组情况，可以动态地连接和判断连接，广泛应用于网络连接、图的连通分量、最小生成树等问题。

核心思想

并查集通过数组或者特殊结构存储每个元素的父节点信息。初始时，每个元素的父节点是其自身，表示每个元素自成一个集合。通过路径压缩和按秩合并等优化策略，可以提高并查集的效率。

- **路径压缩**：在执行Find操作时，使得路径上的所有点直接指向根节点，这样可以减少后续操作的时间复杂度。
- **按秩合并**：在执行Union操作时，总是将较小的树连接到较大的树的根节点上，这样可以避免树过深，影响操作效率。

代码示例

```
class UnionFind:
    # 初始化
    def __init__(self, size):
        # 将每个节点的上级设置为自己
        self.parent = list(range(size))
        # 每个节点的秩都是0
        self.rank = [0] * size

    # 查找
    def find(self, p):
        if self.parent[p] != p:
```

```
            # 这一步进行了路径压缩。
            # 如果不进行路径压缩，这一步是 return self.find(self.parent[p])
            self.parent[p] = self.find(self.parent[p])
        return self.parent[p]

    # 合并
    def union(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        if rootP != rootQ:
            # 按秩合并，总是将较小的树连接到较大的树的根节点上
            if self.rank[rootP] > self.rank[rootQ]:
                self.parent[rootQ] = rootP
            elif self.rank[rootP] < self.rank[rootQ]:
                self.parent[rootP] = rootQ
            else:
                # 如果两个节点的秩相等，就无所谓
                self.parent[rootQ] = rootP
                # 但这时需要把连接后较大的节点的秩+1
                self.rank[rootP] += 1

    # 是否属于同一集合
    def connected(self, p, q):
        return self.find(p) == self.find(q)
```

例题

OJ02524:宗教信仰

http://cs101.openjudge.cn/dsapre/02524/

最基本的应用，只是最后多了一步看看有多少个集合。

```
class UnionFind:
    def __init__(self, size):
        self.parent = [i for i in range(size + 1)]
        self.rank = [0] * (size + 1)

    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):
        x_parent = self.find(x)
        y_parent = self.find(y)
        if x_parent != y_parent:
            if self.rank[x_parent] > self.rank[y_parent]:
                self.parent[y_parent] = x_parent
            elif self.rank[x_parent] < self.rank[y_parent]:
                self.parent[x_parent] = y_parent
            else:
```

```
                self.parent[y_parent] = x_parent
                self.rank[x_parent] += 1


n_case = 0
while True:
    n_case += 1
    n, m = map(int, input().split())
    if m == 0 and n == 0:
        break
    uf = UnionFind(n)
    for i in range(m):
        a, b = map(int, input().split())
        uf.union(a, b)
    cnt = set([uf.find(i) for i in uf.parent])  # 这一步是多的
    print(f'Case {n_case}:', len(cnt) - 1)
```

OJ18250:冰阔落 I

http://cs101.openjudge.cn/2024sp_routine/18250/ **描述** 老王喜欢喝冰阔落。

初始时刻，桌面上有n杯阔落，编号为1到n。老王总想把其中一杯阔落倒到另一杯中，这样他一次性就能喝很多很多阔落，假设杯子的容量是足够大的。

有m 次操作，每次操作包含两个整数x与y。

若原始编号为x 的阔落与原始编号为y的阔落已经在同一杯，请输出"Yes"；否则，我们将原始编号为y 所在杯子的所有阔落，倒往原始编号为x 所在的杯子，并输出"No"。

最后，老王想知道哪些杯子有冰阔落。

输入 有多组测试数据，少于 5 组。 每组测试数据，第一行两个整数 n, m (n, m<=50000)。接下来 m 行，每行两个整数 x, y (1<=x, y<=n)。 **输出** 每组测试数据，前 m 行输出 "Yes" 或者 "No"。 第 m+1 行输出一个整数，表示有阔落的杯子数量。 第 m+2 行有若干个整数，从小到大输出这些杯子的编号。 **样例输入** 3 2 1 2 2 1 4 2 1 2 4 3 **样例输出** No Yes 2 1 3 No No 2 1 4 这题一开始WA，后来检查，发现原因是按秩合并时，parent[x]不一定更新了。虽然最后用self.find(x)又压缩了一次，仍然可能指向的不是最深的节点。好在此题数据小，无需按秩合并。

```
class DJS:
    def __init__(self, size):
        self.parent = [i for i in range(size + 1)]
        self.rank = [0 for _ in range(size + 1)]

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

    def union(self, a, b):
        root_a = self.find(a)
        root_b = self.find(b)
```

```
            self.parent[root_b] = root_a
            """
            if root_b != root_a:
                if self.rank[root_b] == self.rank[root_a]:
                    self.parent[root_b] = root_a
                    self.rank[root_a] += 1
                elif self.rank[root_b] > self.rank[root_a]:
                    self.parent[root_a] = root_b
                else:
                    self.parent[root_b] = root_a
            """

        def check(self, a, b):
            if self.find(a) == self.find(b):
                print('Yes')
            else:
                print('No')


    while True:
        try:
            n, m = map(int, input().split())
        except EOFError:
            break
        d = DJS(n)
        for _ in range(m):
            x, y = map(int, input().split())
            d.check(x, y)
            d.union(x, y)
        cnt = 0
        ans = []
        for i in range(1, n + 1):
            if d.find(i) == i:
                cnt += 1
                ans.append(i)
        print(len(ans))
        print(*ans)
```

Trie（字典树）

实现

```
class trie:
    def __init__(self):
        self.nex = [[0 for i in range(26)] for j in range(100000)]
        self.cnt = 0
        self.exist = [False] * 100000  # 该结点结尾的字符串是否存在

    def insert(self, s):  # 插入字符串
        p = 0
```

```
        for i in s:
            c = ord(i) - ord("a")
            if not self.nex[p][c]:
                self.cnt += 1
                self.nex[p][c] = self.cnt  # 如果没有，就添加结点
            p = self.nex[p][c]
        self.exist[p] = True

    def find(self, s):  # 查找字符串
        p = 0
        for i in s:
            c = ord(i) - ord("a")
            if not self.nex[p][c]:
                return False
            p = self.nex[p][c]
        return self.exist[p]
```

OJ04089:电话号码

http://bailian.openjudge.cn/practice/4089/ **描述** 给你一些电话号码，请判断它们是否是一致的，即是否有某个电话是另一个电话的前缀。比如：

Emergency 911 Alice 97 625 999 Bob 91 12 54 26

在这个例子中，我们不可能拨通Bob的电话，因为Emergency的电话是它的前缀，当拨打Bob的电话时会先接通Emergency，所以这些电话号码不是一致的。

输入 第一行是一个整数t，1 ≤ t ≤ 40，表示测试数据的数目。 每个测试样例的第一行是一个整数n，1 ≤ n ≤ 10000，其后n行每行是一个不超过10位的电话号码。 **输出** 对于每个测试数据，如果是一致的输出“YES”，如果不是输出“NO”。 **样例输入** 2 3 911 97625999 91125426 5 113 12340 123440 12345 98346 **样例输出** NO YES

```
def build_trie(s, parent: dict):
    if s[0] in parent.keys():
        if len(s) == 1:
            return False
        if not parent[s[0]]:
            return False
        return build_trie(s[1:], parent[s[0]])
    parent[s[0]] = {}
    if len(s) == 1:
        return True
    return build_trie(s[1:], parent[s[0]])
t = int(input())
for i in range(t):
    trie = {}
    n = int(input())
    flag = False
    for j in range(n):
        number = input()
        if flag:
```

```
            continue
        if not build_trie(number, trie):
            print('NO')
            flag = True
    if not flag:
        print('YES')
```

堆

堆（Heap）是一种特别的完全二叉树。所有的节点都大于等于（最大堆）或小于等于（最小堆）每个它的子节点。本文将主要讨论最小堆的实现，其中每个父节点的值都小于或等于其子节点的值。

最小堆的实现原理

最小堆通常可以用一个数组来实现，利用数组的索引来模拟树结构：

- 根节点位于数组的第一个位置，即 `index = 0`。
- 对于任意位于index = i的节点：
  - 其左子节点的位置是 `2 * i + 1`
  - 其右子节点的位置是 `2 * i + 2`
  - 其父节点的位置是 `(i - 1) / 2`（这里的除法为整数除法）

最小堆的核心操作

1. **插入操作（Add）**
   - 将新元素添加到数组的末尾。
   - 从这个新元素开始，向上调整堆，以保持最小堆的性质。这通常被称为“上浮”（bubble up或percolate up），即如果添加的元素小于其父节点，则与父节点交换位置，重复这一过程直到恢复堆的性质或者该节点成为根节点。
2. **删除最小元素（Extract Min）**
   - 最小元素总是位于数组的第一个位置。
   - 将数组最后一个元素移动到第一个位置，然后从根节点开始向下调整堆（下沉或percolate down）。如果父节点大于任一子节点，则与最小的子节点交换位置，重复这一过程直到恢复堆的性质或者该节点成为叶节点。
3. **获取最小元素（Find Min）**
   - 由于最小元素总是位于数组的第一位置，因此获取最小元素非常高效，时间复杂度为 O(1)。
4. **堆化（Heapify）**
   - 将一个不满足最小堆性质的数组转换成最小堆。这通常通过从最后一个非叶子节点开始，依次对每个节点执行“下沉”操作来实现。非叶子节点的开始位置可以从 `n/2 - 1` 开始（`n` 是数组长度），这是因为所有更后面的节点都是叶子节点，已经满足堆的性质。

性能

- 插入和删除操作的时间复杂度通常是 O(log n)，因为需要在树的高度上进行操作（上浮或下沉），而树的高度与节点数的对数成正比。
- 堆化操作的时间复杂度是 O(n)，这是通过精心构造的下沉操作实现的，虽然看起来每个节点都要处理，但实际上更深的节点较少，处理起来也更快。

OJ04078:实现堆结构

http://cs101.openjudge.cn/practice/04078/ **描述** 定义一个数组，初始化为空。在数组上执行两种操作： 1、增添1个元素，把1个新的元素放入数组。 2、输出并删除数组中最小的数。 使用堆结构实现上述功能的高效算法。

输入 第一行输入一个整数n，代表操作的次数。 每次操作首先输入一个整数type。 当type=1，增添操作，接着输入一个整数u，代表要插入的元素。 当type=2，输出删除操作，输出并删除数组中最小的元素。 1<=n<=100000。 **输出** 每次删除操作输出被删除的数字。 **样例输入** 4 1 5 1 1 1 7 2 **样例输出** 1 **提示** 每组测试数据的复杂度为O(nlogn)的算法才能通过本次，否则会返回TLE(超时) 需要使用最小堆结构来实现本题的算法

```
from math import floor as floor


class MinHeap:
    def __init__(self):
        self.value = []

    def get_min(self):
        if not self.value:
            return None
        return self.value[0]

    def swap(self, a, b):
        self.value[a], self.value[b] = self.value[b], self.value[a]

    def insert(self, x):
        self.value.append(x)
        index = len(self.value) - 1
        while True:
            parent_index = floor((index - 1) / 2)
            if index <= 0 or self.value[parent_index] <=
self.value[index]:
                return
            self.swap(index, parent_index)
            index = parent_index

    def delete_min(self):
        if not self.value:
            return
        self.swap(0, -1)
        self.value.pop()
        index = 0
        while index < len(self.value):
            left_index, right_index = 2 * index + 1, 2 * index + 2
            if left_index >= len(self.value):
                return
            if right_index >= len(self.value):
                if self.value[index] > self.value[left_index]:
                    self.swap(index, left_index)
                    continue
                else:
                    return
            if self.value[index] < min(self.value[left_index],
```

```
self.value[right_index]):
                return
            small = left_index if self.value[left_index] <
self.value[right_index] else right_index
            self.swap(small, index)
            index = small


heap = MinHeap()
for i in range(int(input())):
    s = input()
    if s[0] == '1':
        heap.insert(int(s[2:]))
    else:
        print(heap.get_min())
        heap.delete_min()
```

Huffman树

OJ22161:哈夫曼编码树

http://cs101.openjudge.cn/practice/22161/ **描述** 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 **输入** 第一行是整数n，表示字符集有n个字符。 接下来n行，每行是一个字符及其使用频率（权重）。字符都是英文字母。 再接下来是若干行，有的是字母串，有的是01编码串。 **输出** 对输入中的字母串，输出该字符串的编码 对输入中的01串,将其解码，输出原始字符串 **样例输入** 3 g 4 d 8 c 10 dc 110 **样例输出** 110 dc **提示** 数据规模很小，不用在乎效率

```
import heapq


class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
        self.height = None

    def huff_value(self, h):
        if not self.left and not self.right:
            return h * self.val
        left_value, right_value = 0, 0
        if self.left:
            left_value = self.left.huff_value(h + 1)
        if self.right:
```

```
            right_value = self.right.huff_value(h + 1)
        return left_value + right_value

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

    def __gt__(self, other):
        return self.val > other.val


n = int(input())
nodes = []
for i in list(map(int, input().split())):
    heapq.heappush(nodes, Node(i))
while len(nodes) > 1:
    left, right = heapq.heappop(nodes), heapq.heappop(nodes)
    heapq.heappush(nodes, Node(left.val + right.val, left, right))
print(nodes[0].huff_value(0))
```

```
import heapq
class Node:
    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 build_huffman_tree(characters):
    heap = []
    for char, weight in characters.items():
        heapq.heappush(heap, Node(weight, char))
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        #merged = Node(left.weight + right.weight) #note: 合并后，char 字段默
认值是空
        merged = Node(left.weight + right.weight, min(left.char,
right.char))
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
    return heap[0]
def encode_huffman_tree(root):
    codes = {}
    def traverse(node, code):
        #if node.char:
        if node.left is None and node.right is None:
```

```
            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.left is None and node.right is None:
            decoded += node.char
            node = root
    return decoded

# 读取输入
n = int(input())
characters = {}
for _ in range(n):
    char, weight = input().split()
    characters[char] = int(weight)
# 构建哈夫曼编码树
huffman_tree = build_huffman_tree(characters)
# 编码和解码
codes = encode_huffman_tree(huffman_tree)
strings = []
while True:
    try:
        line = input()
        strings.append(line)
    except EOFError:
        break
results = []
for string in strings:
    if string[0] in ('0','1'):
        results.append(huffman_decoding(huffman_tree, string))
    else:
        results.append(huffman_encoding(codes, string))
for result in results:
    print(result)
```

AVL树

```
from search.BST import DictBST as BST
from search.BST import Assoc
from search.AVLNode import AVLNode


class AVL(BST):
    def __init__(self):
        BST.__init__(self)
        pass

    def insert(self, key, value):
        # a指向离插入位置最近的BF非0的节点，p是扫描变量，p从a的子节点开始遍历，用于更新BF，pa指向a的父节点,q是插入节点的父节点
        a = p = self._root
        if a is None:
            self._root = AVLNode(Assoc(key, value))
            return
        pa = q = None
        while p is not None:  # 确定插入位置的最小非平衡子树，也就是找出a的指向
            if key == p.data.key():  # key存在，更新值结束
                p.data.setValue(value)
                return
            if p.bf != 0:
                pa, a = q, p  # 已知最小非平衡树
            # 把q指向p的也就是指向插入位置的父节点
            q = p
            if key < p.data.key():
                p = p.left
            else:
                p = p.right
        node = AVLNode(Assoc(key, value))
        if key < q.data.key():
            q.left = node  # 作为左节点
        else:
            q.right = node  # 作为右节点
        # 新节点已经插入，a是最小不平衡子树
        if key < a.data.key():
            p = b = a.left
            d = 1  # d记录新节点在a的哪颗子树
        else:
            p = b = a.right
            d = -1
        # 修改b到新节点路径上各节点的BF值,b为a的子节点
        while p != node:
            if key < p.data.key():  # p的左子树增高
                p.bf = 1
                p = p.left
            else:  # p的右子树增高
                p.bf = -1
                p = p.right
        if a.bf == 0:  # a的原BF为0，不会失衡
            a.bf = d
            return
```

```
            if a.bf == -d:  # 新节点在较低的子树中
                a.bf = 0
                return
            # 新节点在较高的子树中，发生失衡，需要调整
            if d == 1:
                if b.bf == 1:
                    b = AVL.LL(a, b)
                else:
                    b = AVL.LR(a, b)
            else:
                if b.bf == -1:
                    b = AVL.RR(a, b)
                else:
                    b = AVL.RL(a, b)

            if pa is None:  # 原a为树根，修改根
                self._root = b
            else:
                if pa.left == a:
                    pa.left = b
                else:
                    pa.right = b

        # LL型调整，右旋，踢掉左孩子的右孩子，被踢掉的孩子变成旋转节点的左孩子，返回调整后的节点
        # a为根节点，b为a的孩子节点，下同
        @staticmethod
        def LL(a, b):
            print("LL型调整 %s结点与%s结点 " % (str(a.data.key()), str(b.data.key())))
            a.left = b.right
            b.right = a
            a.bf = b.bf = 0
            return b

        # RR型调整，左旋
        @staticmethod
        def RR(a, b):
            print("RR型调整 %s结点与%s结点 " % (str(a.data.key()), str(b.data.key())))
            a.right = b.left
            b.left = a
            a.bf = b.bf = 0
            return b

        # LR型调整，先左旋再右旋
        @staticmethod
        def LR(a, b):
            print("LR型调整 %s结点与%s结点 " % (str(a.data.key()), str(b.data.key())))
            c = b.right  # 找到根节点的左子树的右子树
            # 左旋和右旋
            a.left, b.right = c.right, c.left
            c.left, c.right = b, a
```

```
            if c.bf == 0:  # 本身就是新插入的节点
                a.bf = b.bf = 0
            elif c.bf == 1:  # 新节点为c的左子树
                a.bf = -1
                b.bf = 0
            else:  # 新节点为c的右子树
                a.bf = 0
                b.bf = 1
            c.bf = 0  # 实现平衡
            return c

        # RL型调整，先右旋再左旋
        @staticmethod
        def RL(a, b):
            print("RL型调整 %s结点与%s结点 " % (str(a.data.key()),
    str(b.data.key())))
            c = b.left  # 找到根节点的右子树的左子树
            a.right, b.left = c.left, c.right
            c.left, c.right = a, b
            if c.bf == 0:
                a.bf = 0
                b.bf = 0
            elif c.bf == 1:
                a.bf = 0
                b.bf = -1
            else:
                a.bf = -1
                b.bf = 0
            c.bf = 0
            return c


    if __name__ == '__main__':
        tree = AVL()
        tree.insert(57, "57")
        tree.insert(36, "36")
        tree.insert(23, "23")
        tree.insert(11, "11")
        tree.insert(18, "18")
        tree.insert(69, "69")
        tree.insert(81, "81")
        tree.insert(63, "63")
        tree.insert(60, "60")
        print("中序遍历")
        tree.printValues()
        pass
```

图

BFS与DFS

标准代码

```
graph = {
    "A": ["B", "C"],
    "B": ["A", "C", "D"],
    "C": ["A", "B", "D", "E"],
    "D": ["B", "C", "E", "F"],
    "E": ["C", "D"],
    "F": ["D"],
}
def BFS(graph, s):
    queue = []
    queue.append(s)
    seen = set()
    seen.add(s)
    parent = {s:None
    }
    while (len(queue) > 0):
        vertex = queue.pop(0)
        nodes = graph[vertex]
        for w in nodes:
            if w not in seen:
                queue.append(w)
                seen.add(w)
                parent[w] = vertex
        print(vertex)
    return parent

parent = BFS(graph, "E")
v = 'B'
while v!= None:
    print(v)
    v = parent[v]
```

```
graph = {
    "A": ["B", "C"],
    "B": ["A", "C", "D"],
    "C": ["A", "B", "D", "E"],
    "D": ["B", "C", "E", "F"],
    "E": ["C", "D"],
    "F": ["D"],
}
def DFS(graph, s):
    stack = []
    stack.append(s)
    seen = set()
    seen.add(s)
    while (len(stack) > 0):
            vertex = stack.pop()
            nodes = graph[vertex]
```

```
            for w in nodes:
                if w not in seen:
                    stack.append(w)
                    seen.add(w)
            print(vertex)
```

28046:词梯

http://cs101.openjudge.cn/practice/28046/ **描述** 词梯问题是由“爱丽丝漫游奇境”的作者 Lewis Carroll 在1878年所发明的单词游戏。从一个单词演变到另一个单词，其中的过程可以经过多个中间单词。要求是相邻两个单词之间差异只能是1个字母，如fool -> pool -> poll -> pole -> pale -> sale -> sage。与“最小编辑距离”问题的区别是，中间状态必须是单词。目标是找到最短的单词变换序列。

假设有一个大的单词集合（或者全是大写单词，或者全是小写单词），集合中每个元素都是四个字母的单词。采用图来解决这个问题，如果两个单词的区别仅在于有一个不同的字母，就用一条边将它们相连。如果能创建这样一个图，那么其中的任意一条连接两个单词的路径就是词梯问题的一个解，我们要找最短路径的解。下图展示了一个小型图，可用于解决从 fool 到sage的词梯问题。

注意，它是无向图，并且边没有权重。

输入 输入第一行是个正整数 n，表示接下来有n个四字母的单词，每个单词一行。2 <= n <= 4000。 随后是 1 行，描述了一组要找词梯的起始单词和结束单词，空格隔开。 **输出** 输出词梯对应的单词路径，空格隔开。如果不存在输出 NO。 如果有路径，保证有唯一解。 **样例输入** 25 bane bank bunk cane dale dunk foil fool kale lane male mane pale pole poll pool quip quit rain sage sale same tank vain wane fool sage **样例输出** fool pool poll pole pale sale sage

```
from collections import deque

def is_one_letter_diff(word1, word2):
    diff_count = 0
    for a, b in zip(word1, word2):
        if a != b:
            diff_count += 1
        if diff_count > 1:
            return False
    return diff_count == 1

def find_word_ladder(word_list, start, end):
    word_set = set(word_list)
    if end not in word_set:
        return ["NO"]

    queue = deque([start])
    parent = {start: None}

    while queue:
        current_word = queue.popleft()

        if current_word == end:
            path = []
```

```
            while current_word:
                path.append(current_word)
                current_word = parent[current_word]
            return path[::-1]

        for word in list(word_set):
            if is_one_letter_diff(current_word, word) and word not in
parent:
                parent[word] = current_word
                queue.append(word)
                word_set.remove(word)

    return ["NO"]

def main():
    n = int(input())
    word_list = [input().strip() for _ in range(n)]
    start, end = input().strip().split()

    result = find_word_ladder(word_list, start, end)

    print(" ".join(result))

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

28050:骑士周游 **描述** 在一个国际象棋棋盘上，一个棋子“马”（骑士），按照“马走日”的规则，从一个格子出发，要走遍所有棋盘格恰好一次。把一个这样的走棋序列称为一次“周游”。在 8 × 8 的国际象棋棋盘上，合格的“周游”数量有 1.305×10^35这么多，走棋过程中失败的周游就更多了。

采用图搜索算法，是解决骑士周游问题最容易理解和编程的方案之一，解决方案分为两步： 首先用图表示骑士在棋盘上的合理走法； 采用图搜索算法搜寻一个长度为（行 × 列-1）的路径，路径上包含每个顶点恰一次。

输入 两行。 第一行是一个整数n，表示正方形棋盘边长，3 <= n <= 19。

第二行是空格分隔的两个整数sr, sc，表示骑士的起始位置坐标。棋盘左上角坐标是 0 0。0 <= sr <= n-1, 0 <= sc <= n-1。 **输出** 如果是合格的周游，输出 success，否则输出 fail。 **样例输入** 5 0 0 **样例输出** success

```
import sys
sys.setrecursionlimit(10**7)

MOVES = [(-2, -1), (-2, 1), (-1, -2), (-1, 2),
         ( 1, -2), ( 1, 2), ( 2, -1), ( 2, 1)]

def knights_tour_exists(n, sr, sc):
    visited = [[False]*n for _ in range(n)]
    total = n*n

    def dfs(r, c, count):
        if count == total:
```

```
                return True
            nbrs = []
            for dr, dc in MOVES:
                nr, nc = r+dr, c+dc
                if 0 <= nr < n and 0 <= nc < n and not visited[nr][nc]:
                    deg = 0
                    for dr2, dc2 in MOVES:
                        r2, c2 = nr+dr2, nc+dc2
                        if 0 <= r2 < n and 0 <= c2 < n and not visited[r2]
[c2]:
                            deg += 1
                    nbrs.append((deg, nr, nc))
            nbrs.sort(key=lambda x: x[0])

            for _, nr, nc in nbrs:
                visited[nr][nc] = True
                if dfs(nr, nc, count+1):
                    return True
                visited[nr][nc] = False
            return False

    visited[sr][sc] = True
    return dfs(sr, sc, 1)

def main():
    data = sys.stdin.read().split()
    n = int(data[0])
    sr, sc = map(int, data[1:3])
    if knights_tour_exists(n, sr, sc):
        print("success")
    else:
        print("fail")

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

拓扑排序及Kahn算法

拓扑排序是对有向无环图（DAG，Directed Acyclic Graph）的顶点进行排序的一种方法，使得对于图中的每条有向边 UV（从顶点 U 指向顶点 V），U 在排序中都出现在 V 之前。拓扑排序不是唯一的，一个有向无环图可能有多个有效的拓扑排序。

拓扑排序常用的算法包括基于 DFS（深度优先搜索）的方法和基于 BFS（广度优先搜索，也称为Kahn算法）的方法。

作用：检测是否有环

代码示例

```
from collections import deque, defaultdict
```

```
def topological_sort(vertices, edges):
    # 计算所有顶点的入度
    in_degree = {v: 0 for v in vertices}
    graph = defaultdict(list)

    # u->v
    for u, v in edges:
        graph[u].append(v)
        in_degree[v] += 1  # v的入度+1

    # 将所有入度为0的顶点加入队列
    queue = deque([v for v in vertices if in_degree[v] == 0])
    sorted_order = []

    while queue:
        u = queue.popleft()
        sorted_order.append(u)

        # 对于每一个相邻顶点，减少其入度
        for v in graph[u]:
            in_degree[v] -= 1
            # 如果入度减为0，则加入队列
            if in_degree[v] == 0:
                queue.append(v)

    if len(sorted_order) != len(vertices):
        return None  # 存在环，无法进行拓扑排序
    return sorted_order


# 示例使用
vertices = ['A', 'B', 'C', 'D', 'E', 'F']
edges = [('A', 'D'), ('F', 'B'), ('B', 'D'), ('F', 'A'), ('D', 'C')]
result = topological_sort(vertices, edges)
if result:
    print("拓扑排序结果:", result)
else:
    print("图中有环，无法进行拓扑排序")
```

例题

OJ04084:拓扑排序

http://cs101.openjudge.cn/2024sp_routine/04084/ **描述** 给出一个图的结构，输出其拓扑排序序列，要求在同等条件下，编号小的顶点在前。 **输入** 若干行整数，第一行有2个数，分别为顶点数v和弧数a，接下来有a行，每一行有2个数，分别是该条弧所关联的两个顶点编号。 v<=100, a<=500 **输出** 若干个空格隔开的顶点构成的序列(用小写字母)。 **样例输入** 6 8 1 2 1 3 1 4 3 2 3 5 4 5 6 4 6 5 **样例输出** v1 v3 v2 v6 v4 v5 拓扑排序，但是要求“同等条件下，编号小的顶点在前”，不得不把普通队列转换成一个优先队列了。

```
from collections import deque, defaultdict
import heapq

def topo_sort(g, nv):
    ans = []
    deg = {v: 0 for v in range(1, nv+1)}
    child = {v: [] for v in range(1, nv+1)}
    for u, v in g:
        # u->v
        if v not in deg:
            deg[v] = 1
        else:
            deg[v] += 1
        if u not in child:
            child[u] = [v]
        else:
            child[u].append(v)
    q = [v for v in deg.keys() if deg[v] == 0]
    heapq.heapify(q)
    while q:
        now = heapq.heappop(q)
        ans.append(now)
        for i in child[now]:
            deg[i] -= 1
            if deg[i] == 0:
                heapq.heappush(q, i)

    return ans
v, a = map(int, input().split())
g = []
for _ in range(a):
    x, y = map(int, input().split())
    g.append([x, y])
for i in topo_sort(g, v):
    print('v' + str(i), end=' ')
```

OJ09202:舰队、海域出击!

http://cs101.openjudge.cn/2024sp_routine/09202/

检测有向图有没有环，拓扑排序，也就是Kahn算法。 **描述** 作为一名海军提督，Pachi将指挥一支舰队向既定海域出击！ Pachi已经得到了海域的地图，地图上标识了一些既定目标和它们之间的一些单向航线。如果我们把既定目标看作点、航线看作边，那么海域就是一张有向图。不幸的是，Pachi是一个会迷路的提督QAQ，所以他在包含环(圈)的海域中必须小心谨慎，而在无环的海域中则可以大展身手。 受限于战时的消息传递方式，海域的地图只能以若干整数构成的数据的形式给出。作为舰队的通讯员，在出击之前，请你告诉提督海域中是否包含环。 **输入** 每个测试点包含多组数据，每组数据代表一片海域，各组数据之间无关。 第一行是数据组数T。 每组数据的第一行两个整数N，M，表示海域中既定目标数、航线数。 接下来M行每行2个不相等的整数x,y，表示从既定目标x到y有一条单向航线（所有既定目标使用1~N的整数表示）。 描述中的图片仅供参考，其顶点标记方式与本题数据无关。 1<=N<=100000，1<=M<=500000，1<=T<=5 注意：输入的有向

图不一定是连通的。 **输出** 输出包含T行。 对于每组数据，输出Yes表示海域有环，输出No表示无环。 样例输入 2 7 6 1 2 1 3 2 4 2 5 3 6 3 7 12 13 1 2 2 3 2 4 3 5 5 6 4 6 6 7 7 8 8 4 7 9 9 10 10 11 10 12 样例输出 No Yes

```
from collections import deque
def topo(g, deg, v):
    q = deque([x for x in deg.keys() if deg[x] == 0])
    cnt = 0
    while q:
        now = q.popleft()
        cnt += 1
        for next in g[now]:
            deg[next] -= 1
            if deg[next] == 0:
                q.append(next)
    if cnt == v:
        print('No')
    else:
        print('Yes')
for _ in range(int(input())):
    v, m = map(int, input().split())
    g = {a: [] for a in range(1, v + 1)}
    deg = {a: 0 for a in range(1, v + 1)}
    for _ in range(m):
        x, y = map(int, input().split())
        deg[y] += 1
        g[x].append(y)
    topo(g, deg, v)
```

27635:判断无向图是否连通有无回路(同23163)

http://cs101.openjudge.cn/practice/27635/ **描述** 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

```
import sys
from collections import deque

def main():
    data = sys.stdin.read().split()
    if not data:
        return

    n = int(data[0])
    m = int(data[1])
    graph = [[] for _ in range(n)]
    index = 2
    for _ in range(m):
```

```
            u = int(data[index])
            v = int(data[index + 1])
            index += 2
            graph[u].append(v)
            graph[v].append(u)

        visited = [False] * n
        # 连通性判断
        count = 0
        queue = deque([0])
        visited[0] = True
        while queue:
            u = queue.popleft()
            count += 1
            for v in graph[u]:
                if not visited[v]:
                    visited[v] = True
                    queue.append(v)

        connected = "yes" if count == n else "no"

        # 重置visited用于回路判断
        visited = [False] * n
        has_loop = False
        parent = [-1] * n

        for i in range(n):
            if not visited[i]:
                stack = [i]
                visited[i] = True
                parent[i] = -1
                while stack:
                    u = stack.pop()
                    for v in graph[u]:
                        if not visited[v]:
                            visited[v] = True
                            parent[v] = u
                            stack.append(v)
                        elif parent[u] != v:
                            has_loop = True

        loop = "yes" if has_loop else "no"

        print(f"connected:{connected}")
        print(f"loop:{loop}")

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

Dijkstra算法

```
import heapq
import math
# 构造数据,邻接表
graph = {
    "A": {"B": 5, "C": 1},
    "B": {"A": 5, "C": 2, "D": 1},
    "C": {"A": 1, "B": 2, "D": 4, "E": 8},
    "D": {"B": 1, "C": 4, "E": 3, "F": 6},
    "E": {"C": 8, "D": 3},
    "F": {"D": 6},
}


def init_distance(graph, s):
    distance = {s: 0}
    for vertex in graph:
        if vertex != s:
            distance[vertex] = math.inf
    return distance


def dijkstra(graph, s):
    pqueue = []
    heapq.heappush(pqueue, (0, s))
    seen = set()
    # 最小生成树
    parent = {s: None}
    distance = init_distance(graph, s)
    while len(pqueue) > 0:
        pair = heapq.heappop(pqueue)
        dist = pair[0]
        vertex = pair[1]
        seen.add(vertex)
        # 放入邻接点
        nodes = graph[vertex].keys()
        for node in nodes:
            if node not in seen:
                if dist + graph[vertex][node] < distance[node]:
                    heapq.heappush(pqueue, (dist + graph[vertex][node],
node))
                    parent[node] = vertex
                    distance[node] = dist + graph[vertex][node]
        # 输出遍历结果
        # print(vertex)
    return parent, distance
    pass


def getMinPath(graph, s, t):
    parent, distance = dijkstra(graph, s)
    dis = distance[t]
    path = ""
    while t != None:
```

```
            path += t + " "
            t = parent[t]
        return path, dis


parent, distance = dijkstra(graph, "A")
print(parent)
print(distance)

print('E到A的最短路径：')
path, dis = getMinPath(graph, 'A', 'E')
print(path)
print("长度为：", dis)
```

例题

走山路

http://cs101.openjudge.cn/practice/20106/ **描述** 某同学在一处山地里，地面起伏很大，他想从一个地方走到另一个地方，并且希望能尽量走平路。 现有一个m*n的地形图，图上是数字代表该位置的高度，"#"代表该位置不可以经过。 该同学每一次只能向上下左右移动，每次移动消耗的体力为移动前后该同学所处高度的差的绝对值。现在给出该同学出发的地点和目的地，需要你求出他最少要消耗多少体力。

输入 第一行是整数 m,n,p，m是行数，n是列数，p是测试数据组数。 0 <= m,n,p <= 100 接下来m行是地形图 再接下来n行每行前两个数是出发点坐标（前面是行，后面是列），后面两个数是目的地坐标（前面是行，后面是列）（出发点、目的地可以是任何地方，出发点和目的地如果有一个或两个在"#"处，则将被认为是无法达到目的地） **输出** n行，每一行为对应的所需最小体力，若无法达到，则输出"NO" **样例输入** 4 5 3 0 0 0 0 0 0 1 1 2 3 ·# 1 0 0 0 0 # 0 0 0 0 0 3 4 1 0 1 4 3 4 3 0 **样例输出** 2 3 NO

解释： 第一组：从左上角到右下角，要上1再下来，所需体力为2 第二组：一直往右走，高度从0变为1，再变为2，再变为3，消耗体力为3 第三组：左下角周围都是"#"，不可以经过，因此到不了 提示 Dijkstra

```
import heapq
def bfs(x1,y1):
    q=[(0,x1,y1)]
    v=set()
    while q:
        t,x,y=heapq.heappop(q)
        v.add((x,y))
        if x==x2 and y==y2:
            return t
        for dx,dy in dir:
            nx,ny=x+dx,y+dy
            if 0<=nx<m and 0<=ny<n and ma[nx][ny]!='#' and (nx,ny) not in
v:
                nt=t+abs(int(ma[nx][ny])-int(ma[x][y]))
                heapq.heappush(q,(nt,nx,ny))
    return 'NO'
m,n,p=map(int,input().split())
ma=[list(input().split()) for _ in range(m)]
```

```
dir=[(1,0),(-1,0),(0,1),(0,-1)]
for _ in range(p):
    x1,y1,x2,y2=map(int,input().split())
    if ma[x1][y1]=='#' or ma[x2][y2]=='#':
        print('NO')
        continue
    print(bfs(x1,y1))
```

OJ05443:兔子与樱花

http://cs101.openjudge.cn/dsapre/05443/ **描述** 很久很久之前，森林里住着一群兔子。有一天，兔子们希望去赏樱花，但当他们到了上野公园门口却忘记了带地图。现在兔子们想求助于你来帮他们找到公园里的最短路。 **输入** 输入分为三个部分。 第一个部分有P+1行（P<30），第一行为一个整数P，之后的P行表示上野公园的地点, 字符串长度不超过20。 第二个部分有Q+1行（Q<50），第一行为一个整数Q，之后的Q行每行分别为两个字符串与一个整数，表示这两点有直线的道路，并显示二者之间的矩离（单位为米）。 第三个部分有R+1行（R<20），第一行为一个整数R，之后的R行每行为两个字符串，表示需要求的路线。 **输出** 输出有R行，分别表示每个路线最短的走法。其中两个点之间，用->(矩离)->相隔。 **样例输入** 6 Ginza Sensouji Shinjukugyoen Uenokouen Yoyogikouen Meijishinguu 6 Ginza Sensouji 80 Shinjukugyoen Sensouji 40 Ginza Uenokouen 35 Uenokouen Shinjukugyoen 85 Sensouji Meijishinguu 60 Meijishinguu Yoyogikouen 35 2 Uenokouen Yoyogikouen Meijishinguu Meijishinguu **样例输出** Uenokouen->(35)->Ginza->(80)->Sensouji->(60)->Meijishinguu->(35)->Yoyogikouen Meijishinguu 模板题目，额外的一点是需要记录路径

```
import heapq

def dijkstra(adjacency, start):
    # 初始化，将其余所有顶点到起始点的距离都设为inf（无穷大）
    distances = {vertex: float('inf') for vertex in adjacency}
    # 初始化，所有点的前一步都是None
    previous = {vertex: None for vertex in adjacency}
    # 起点到自身的距离为0
    distances[start] = 0
    # 优先队列
    pq = [(0, start)]

    while pq:
        # 取出优先队列中，目前距离最小的
        current_distance, current_vertex = heapq.heappop(pq)
        # 剪枝，如果优先队列里保存的距离大于目前更新后的距离，则可以跳过
        if current_distance > distances[current_vertex]:
            continue

        # 对当前节点的所有邻居，如果距离更优，将他们放入优先队列中
        for neighbor, weight in adjacency[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                # 这一步用来记录每个节点的前一步
                previous[neighbor] = current_vertex
                heapq.heappush(pq, (distance, neighbor))
```

```
    return distances, previous

def shortest_path_to(adjacency, start, end):
    # 逐步访问每个节点上一步
    distances, previous = dijkstra(adjacency, start)
    path = []
    current = end
    while previous[current] is not None:
        path.insert(0, current)
        current = previous[current]
    path.insert(0, start)
    return path, distances[end]

# Read the input data
P = int(input())
places = {input().strip() for _ in range(P)}

Q = int(input())
graph = {place: {} for place in places}
for _ in range(Q):
    src, dest, dist = input().split()
    dist = int(dist)
    graph[src][dest] = dist
    graph[dest][src] = dist  # Assuming the graph is bidirectional

R = int(input())
requests = [input().split() for _ in range(R)]

# Process each request
for start, end in requests:
    if start == end:
        print(start)
        continue

    path, total_dist = shortest_path_to(graph, start, end)
    output = ""
    for i in range(len(path) - 1):
        output += f"{path[i]}->({graph[path[i]][path[i+1]]})->"
    output += f"{end}"
    print(output)
```

OJ07735:道路

http://cs101.openjudge.cn/practice/07735/ **描述** N个以 1 ... N 标号的城市通过单向的道路相连:。每条道路包含两个参数：道路的长度和需要为该路付的通行费（以金币的数目来表示）

Bob and Alice 过去住在城市 1.在注意到Alice在他们过去喜欢玩的纸牌游戏中作弊后，Bob和她分手了，并且决定搬到城市N。他希望能够尽可能快的到那，但是他囊中羞涩。我们希望能够帮助Bob找到从1到N最短的路径，前提是他能够付的起通行费。 **输入** 第一行包含一个整数K, 0 <= K <= 10000, 代表Bob能够在他路上花费的最大的金币数。第二行包含整数N， 2 <= N <= 100, 指城市的数目。第三行包含整数R, 1 <= R <= 10000, 指路的数目. 接下来的R行，每行具体指定几个整数S, D, L 和 T来说明关于道路的一些情况，这些整数之间通过空格间隔: S is 道路起始城市, 1 <= S <= N D is 道路终点城市, 1 <= D <= N L is 道路长度, 1 <= L <= 100 T is

通行费 (以金币数量形式度量), 0 <= T <=100 注意不同的道路可能有相同的起点和终点。 **输出** 输入结果应该只包括一行，即从城市1到城市N所需要的最小的路径长度（花费不能超过K个金币）。如果这样的路径不存在，结果应该输出-1。 **样例输入** 5 6 7 1 2 2 3 2 4 3 3 3 4 2 4 1 3 4 1 4 6 2 1 3 5 2 0 5 4 3 2 **样例输出** 11 dijkstra，但是有点区别，加入优先队列的条件不是距离更短，而是金币够用，但是优先队列的比较仍然是用距离比的

```
import heapq

k, n, r = int(input()), int(input()), int(input())


def dij(g, s, e):
    dis = {v: float('inf') for v in range(1, n + 1)}
    dis[s] = 0
    q = [(0, s, 0)]
    heapq.heapify(q)
    while q:
        d, now, fee = heapq.heappop(q)
        if now == n:
            return d
        for neighbor, distance, c in g[now]:
            if fee + c <= k:
                dis[neighbor] = distance + d
                heapq.heappush(q, (distance + d, neighbor, fee + c))
    return -1


g = {v: [] for v in range(1, n + 1)}
for _ in range(r):
    s, e, m, j = map(int, input().split())
    g[s].append((e, m, j))
p = dij(g, 1, n)
print(p)
```

最小生成树（Prim算法、Kruskal算法）

两种算法，Prim，Kruskal

求解最小生成树（Minimum Spanning Tree, MST）的问题在图论中非常重要，尤其是在设计和优化网络、路由算法以及集群分析等领域。最小生成树是一个无向图的生成树，它包括图中所有的顶点，并且选取的边的总权重最小。以下是几种常用的求解最小生成树的算法：

1. Kruskal算法

- **基本思想**：Kruskal算法是一种基于边的贪心算法。它的核心思想是按照边的权重从小到大的顺序选择边，但在选择的同时必须保证不形成环。
- 步骤：

    1. 将所有边按权重从小到大排序。

2. 初始化一个空的最小生成树。
3. 依次考虑排序后的每条边，如果加入这条边不会与已选的边形成环（可以用并查集来检测），则将其加入到最小生成树中。
4. 重复上一步，直到最小生成树中含有 *V*–1 条边，其中 *V* 是图中顶点的数量。

- **复杂度**：如果使用优化的并查集，复杂度可以达到 O*(*ElogE) 或 O(*ElogV*)*。

2. Prim算法

- **基本思想**：Prim算法是基于顶点的贪心算法。它从任意顶点开始，逐渐增加边和顶点，直到包括所有顶点。

- 步骤：

1. 选择任意一个顶点作为起始点。
2. 使用一个优先队列来维护可选择的边（根据边的权重）。
3. 从优先队列中选择一条权重最小的边，如果这条边连接的顶点还未被加入最小生成树，则将此顶点及边加入树中。
4. 更新优先队列，重复上述过程，直到所有顶点都被加入。

- **复杂度**：使用优先队列（如二叉堆），复杂度为 O*(*ElogV*)*。

代码实现

```
from collections import defaultdict
from heapq import *

def prim(vertexs, edges, start='A'):
adjacent_dict = defaultdict(list)
for weight, v1, v2 in edges:
adjacent_dict[v1].append((weight, v1, v2))
adjacent_dict[v2].append((weight, v2, v1))

min_tree = []
visited = [start]
adjacent_vertexs_edges = adjacent_dict[start]
heapify(adjacent_vertexs_edges)

while adjacent_vertexs_edges:
weight, v1, v2 = heappop(adjacent_vertexs_edges)
if v2 not in visited:
visited.append(v2)
min_tree.append((weight, v1, v2))
for next_edge in adjacent_dict[v2]:
if next_edge[2] not in visited:
heappush(adjacent_vertexs_edges, next_edge)
return min_tree

# 定义顶点和边
vertices = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
edges = [
(7, 'A', 'B'), (5, 'A', 'D'), (8, 'B', 'C'), (9, 'B', 'D'),
```

```
(7, 'B', 'E'), (5, 'C', 'E'), (15, 'D', 'E'), (6, 'D', 'F'),
(8, 'E', 'F'), (9, 'E', 'G'), (11, 'F', 'G')
]

# 运行Prim算法
print(prim(vertices, edges, start='A'))
```

OJ05442:兔子与星空

http://cs101.openjudge.cn/practice/05442/ **描述** 很久很久以前，森林里住着一群兔子。兔子们无聊的时候就喜欢研究星座。如图所示，天空中已经有了n颗星星，其中有些星星有边相连。兔子们希望删除掉一些边，然后使得保留下的边仍能是n颗星星连通。他们希望计算，保留的边的权值之和最小是多少?

输入 第一行只包含一个表示星星个数的数n，n不大于26，并且这n个星星是由大写字母表里的前n个字母表示。接下来的n-1行是由字母表的前n-1个字母开头。最后一个星星表示的字母不用输入。对于每一行，以每个星星表示的字母开头，然后后面跟着一个数字，表示有多少条边可以从这个星星到后面字母表中的星星。如果k是大于0，表示该行后面会表示k条边的k个数据。每条边的数据是由表示连接到另一端星星的字母和该边的权值组成。权值是正整数的并且小于100。该行的所有数据字段分隔单一空白。该星星网络将始终连接所有的星星。该星星网络将永远不会超过75条边。没有任何一个星星会有超过15条的边连接到其他星星（之前或之后的字母）。在下面的示例输入，数据是与上面的图相一致的。 **输出** 输出是一个整数，表示最小的权值和 **样例输入** 9 A 2 B 12 I 25 B 3 C 10 H 40 I 8 C 2 D 18 G 55 D 1 E 44 E 2 F 60 G 38 F 0 G 1 H 35 H 1 I 35 **样例输出** 216 **提示** 考虑看成最小生成树问题，注意输入表示。

```
import heapq

def prim(graph, start):
    mst = []
    used = set([start])
    edges = [
        (cost, start, to)
        for to, cost in graph[start].items()
    ]
    heapq.heapify(edges)

    while edges:
        cost, frm, to = heapq.heappop(edges)
        if to not in used:
            used.add(to)
            mst.append((frm, to, cost))
            for to_next, cost2 in graph[to].items():
                if to_next not in used:
                    heapq.heappush(edges, (cost2, to, to_next))

    return mst

def solve():
    n = int(input())
    graph = {chr(i+65): {} for i in range(n)}
    for i in range(n-1):
        data = input().split()
```

```
            star = data[0]
            m = int(data[1])
            for j in range(m):
                to_star = data[2+j*2]
                cost = int(data[3+j*2])
                graph[star][to_star] = cost
                graph[to_star][star] = cost
        mst = prim(graph, 'A')
        print(sum(x[2] for x in mst))

    solve()
```

Kurskal

```
    class DisjSet:
        def __init__(self, n):
            self.parent = [i for i in 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):
            xset, yset = self.find(x), self.find(y)
            if self.rank[xset] > self.rank[yset]:
                self.parent[yset] = xset
            else:
                self.parent[xset] = yset
                if self.rank[xset] == self.rank[yset]:
                    self.rank[yset] += 1


    def kruskal(n, edges):
        dset = DisjSet(n)
        edges.sort(key=lambda x: x[2])
        sol = 0
        for u, v, w in edges:
            u, v = ord(u) - 65, ord(v) - 65
            if dset.find(u) != dset.find(v):
                dset.union(u, v)
                sol += w
        if len(set(dset.find(i) for i in range(n))) > 1:
            return -1
        return sol


    n = int(input())
    edges = []
    for _ in range(n - 1):
        arr = input().split()
```

```
    root, m = arr[0], int(arr[1])
    for i in range(m):
        edges.append((root, arr[2 + 2 * i], int(arr[3 + 2 * i])))
print(kruskal(n, edges))
```

强连通分量与Kosaraju算法

是一个用于寻找有向图中所有强连通分量（Strongly Connected Components，SCC）的高效算法。一个强连通分量是最大的子图，其中任何两个顶点都是双向可达的。这个算法的时间复杂度为 $O(V+E)$，其中 V 是顶点数，E 是边数。

算法步骤

Kosaraju's 算法包括以下几个步骤：

第一步：对原图进行深度优先搜索（DFS）

1. 从任意未访问过的顶点开始，对原始图进行一次深度优先搜索。
2. 每次完成一个顶点的DFS遍历后，将该顶点推入一个栈中。这样做的目的是按照完成时间的顺序保存顶点，确保在进行第二次DFS时，我们从一个SCC的源点（或接近源点）开始。

第二步：获取转置图

1. 将原图的所有边反向，得到转置图。转置图的意思是如果原图中有一条从 u 到 v 的有向边，那么在转置图中就有一条从 v 到 u 的边。

第三步：对转置图进行DFS

1. 依次从栈中弹出顶点，如果该顶点未被访问过，就以该顶点为起点，对转置图进行DFS。
2. 每次从一个顶点开始的DFS可以访问到的所有顶点，都属于同一个强连通分量。
3. 记录下每次DFS访问到的所有顶点，它们组成了一个强连通分量。

算法正确性的关键

- 第一次DFS帮助我们了解每个顶点的完成时间。在转置图中，我们按照原图的完成时间逆序（即最晚完成的顶点最先处理）来处理每个顶点，这样可以保证每当开始一个新的DFS时，我们都是从另一个SCC的源点开始的。
- 在转置图中，如果从顶点 u 可以到达顶点 v，那么在原图中，从 v 也能到达 u。因此，第二次DFS实际上是在追踪原图中每个SCC的边界。

代码示例

以下是使用Python实现Kosaraju's算法的基本框架：

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

```python
def dfs_util(graph, v, visited):
    visited[v] = True
    print(v, end=' ')
    for neighbour in graph[v]:
        if not visited[neighbour]:
            dfs_util(graph, neighbour, visited)

def kosaraju(graph, vertices):
    stack = []
    visited = [False] * vertices

    # Step 1: Fill vertices in stack according to their finishing times
    for i in range(vertices):
        if not visited[i]:
            dfs(graph, i, visited, stack)

    # Step 2: Create a reversed graph
    rev_graph = [[] for _ in range(vertices)]
    for v in range(vertices):
        for neighbour in graph[v]:
            rev_graph[neighbour].append(v)

    # Step 3: Process all vertices in order defined by Stack
    visited = [False] * vertices
    while stack:
        v = stack.pop()
        if not visited[v]:
            print("SCC: ", end='')
            dfs_util(rev_graph, v, visited)
            print()

# Example usage
graph = [
    [1],        # 0 -> 1
    [2],        # 1 -> 2
    [0, 3],     # 2 -> 0, 3
    [4],        # 3 -> 4
    [5],        # 4 -> 5
    [3]         # 5 -> 3
]
kosaraju(graph, 6)
```

这段代码首先实现了一个DFS来填充栈，然后创建一个转置图，最后再根据栈中顶点的顺序对转置图执行DFS来找到所有的强连通分量。