DS sheating-sheet

hupxer-7meqqe-hUgzik

工具

int(str,n) 将字符串`str`转换为`n`进制的整数。

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。

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。

math.pow(m,n) 计算`m`的`n`次幂。

math.log(m,n) 计算以`n`为底的`m`的对数。

lrucache

```
from functools import lru_cache
@lru_cache(maxsize=None)
```

bisect

```
import bisect
# 创建一个有序列表
sorted_list = [1, 3, 4, 4, 5, 7]
# 使用bisect_left查找插入点
position = bisect.bisect_left(sorted_list, 4)
print(position)  # 输出: 2
# 使用bisect_right查找插入点
position = bisect.bisect_right(sorted_list, 4)
print(position)  # 输出: 4
# 使用insort_left插入元素
bisect.insort_left(sorted_list, 4)
print(sorted_list)  # 输出: [1, 3, 4, 4, 4, 5, 7]
# 使用insort_right插入元素
```

```python
bisect.insort_right(sorted_list, 4)
print(sorted_list)  # 输出: [1, 3, 4, 4, 4, 4, 5, 7]
```

字符串

1. `str.lstrip()` / `str.rstrip()`: 移除字符串左侧/右侧的空白字符。
2. `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。
3. `str.replace(old, new)`: 将字符串中的old子字符串替换为new。
4. `str.startswith(prefix)` / `str.endswith(suffix)`: 检查字符串是否以prefix开头或以suffix结尾。
5. `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。

   6.`str.title()`：每个单词首字母大写。

counter：计数

```python
from collections import Counter
# 创建一个Counter对象
count = Counter(['apple', 'banana', 'apple', 'orange', 'banana',
'apple'])
# 输出Counter对象
print(count)  # 输出: Counter({'apple': 3, 'banana': 2, 'orange': 1})
# 访问单个元素的计数
print(count['apple'])  # 输出: 3
# 访问不存在的元素返回0
print(count['grape'])  # 输出: 0
# 添加元素
count.update(['grape', 'apple'])
print(count)  # 输出: Counter({'apple': 4, 'banana': 2, 'orange': 1,
'grape': 1})
```

permutations：全排列

```python
from itertools import permutations
# 创建一个可迭代对象的排列
perm = permutations([1, 2, 3])
# 打印所有排列
for p in perm:
    print(p)
# 输出: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)
```

combinations：组合

```
from itertools import combinations
# 创建一个可迭代对象的组合
comb = combinations([1, 2, 3], 2)
# 打印所有组合
for c in comb:
    print(c)
# 输出: (1, 2), (1, 3), (2, 3)
```

reduce：累次运算

```
from functools import reduce
# 使用reduce计算列表元素的乘积
product = reduce(lambda x, y: x * y, [1, 2, 3, 4])
print(product)  # 输出: 24
```

product：笛卡尔积

```
from itertools import product
# 创建两个可迭代对象的笛卡尔积
prod = product([1, 2], ['a', 'b'])
# 打印所有笛卡尔积对
for p in prod:
    print(p)
# 输出: (1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')
```

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
```

namedtuple

`namedtuple` 生成可以使用名字来访问元素内容的元组子类。

```python
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
```

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。

```python
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
```

heapq

1. `heapify(x)`

- **用途**：将列表 x 原地转换为堆。
- 示例

  ```python
  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]
```

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。

```
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'
```

math

gcd包，计算最大公因式

```
from math import gcd
x = gcd(15,20,25)
print(x)
## 5
```

math.pow(m,n)　计算m的n次幂。

math.log(m,n) 计算以n为底的m的对数。

eval

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
所谓表达式就是：eval 这个函数会把里面的字符串参数的引号去掉，把中间的内容当成 Python的代码，eval 函数会执行这段代码并且返回执行结果
也可以这样来理解：eval() 函数就是实现 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
```

埃氏筛法，得到质数表

```
def judge(number):
    nlist = list(range(1,number+1))
    nlist[0] = 0
    k = 2
    while k * k <= number:
        if nlist[k-1] != 0:
            for i in range(2*k,number+1,k):
                nlist[i-1] = 0
        k += 1
    result = []
    for num in nlist:
        if num != 0:
            result.append(num)
    return result
```

print保留小数

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

动态规划

背包问题

```python
# 0-1
dp = [0 for i in range(m+1)]
for i in range(1,n+1):
    for j in range(m,w[i]-1,-1):
        dp[j] = max(dp[j],dp[j-w[i]]+v[i])

# complete
dp = [0 for i in range(m+1)]
for i in range(1,n+1):
    for j in range(w[i],m+1):
        dp[j] = max(dp[j],dp[j-w[i]]+v[i])

# multi
W = [None]
V = [None]

for i in range(1,n+1):
    w,v,num = map(int,input().strip().split())
    k = 1
    while k <= num:
        num = num - k
        W.append(k*w)
        V.append(k*v)
        k = k*2

    if num > 0:
        W.append(num*w)
        V.append(num*v)

dp = [0 for i in range(m+1)]
for i in range(1,len(W)):
    for j in range(m,W[i]-1,-1):
        dp[j] = max(dp[j],dp[j-W[i]]+V[i])
```

最长下降子列

```
# 二分版
from bisect import *

n = int(input())
a = list(map(int, input().split()))
ans = 0
b = [-a[0]]
for i in range(1, n):
    if -a[i] > b[-1]:
        b.append(-a[i])
    else:
        pos = bisect_left(b, -a[i])
        b[pos] = -a[i]
print(len(b))
```

单调栈

维护所有前缀的后缀最值

```
import bisect

n = int(input())
a = [0] * n
for i in range(n):
    a[i] = int(input())

st1 = []
st2 = []
ans = 0
for i in range(n):
    while st1 and a[i] <= a[st1[-1]]:
        st1.pop()
    while st2 and a[i] > a[st2[-1]]:
        st2.pop()
    if st2:
        k = bisect.bisect_right(st1, st2[-1])
        if k < len(st1):
            ans = max(ans, i - st1[k] + 1)
    elif st1:
```

```
            ans = max(ans, i - st1[0] + 1)
        st1.append(i)
        st2.append(i)

print(ans)
```

排序

归并排序

```
def merge_sort(lst):
    # The list is already sorted if it contains a single element.
    if len(lst) <= 1:
        return lst, 0

    # Divide the input into two halves.
    middle = len(lst) // 2
    left, inv_left = merge_sort(lst[:middle])
    right, inv_right = merge_sort(lst[middle:])

    merged, inv_merge = merge(left, right)

    # The total number of inversions is the sum of inversions in the
recursion and the merge process.
    return merged, inv_left + inv_right + inv_merge

def merge(left, right):
    merged = []
    inv_count = 0
    i = j = 0

    # Merge smaller elements first.
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
            inv_count += len(left) - i #left[i~mid)都比right[j]要大，他们都
会与right[j]构成逆序对，将他们加入答案
```

```
    # If there are remaining elements in the left or right half, append them to the result.
    merged += left[i:]
    merged += right[j:]

    return merged, inv_count

while True:
    n = int(input())
    if n == 0:
        break

    lst = []
    for _ in range(n):
        lst.append(int(input()))

    _, inversions = merge_sort(lst)
    print(inversions)
```

树算法

四种遍历

前(中、后）序遍历

```
    def __show(self,root):
        if root!=None:
            print(root.data,end=',')
            self.__show(root.left)
            self.__show(root.right)
        else:
            return 0
```

层次遍历

```
    def level_order(self,root):
        queue = deque()
        queue.append(root)  # 将根节点放到队列之中
        while len(queue) > 0:  # 只要队列不空
            node = queue.popleft()  # 队头出队
            print(node.data, end=',')
            if node.left:  # 出队的节点存在左孩子，就把左孩子入队
                queue.append(node.left)
            if node.right:  # 如果存在右孩子，就把右孩子入队
                queue.append(node.right)
```

建树

current移动法

```
class Node:
    def __init__(self, _v):
        self.v = _v
        self.left = None
        self.right = None
        self.father = None


n = int(input())
for _ in range(n):
    root = Node(input())
    cu = root
    cu_h = 1
    while True:
        s = input()
        if s == '0':
            break
        nn = Node(s[-1])
        while cu_h != len(s) - 1:
            cu = cu.father
            cu_h -= 1
        if cu.left:
            cu.right = nn
        else:
            cu.left = nn
        if s[-1] != '*':
            nn.father = cu
```

```
            cu = nn
            cu_h += 1
```

递归建树

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


index = 0


def build():
    global index
    root = Node(s[index])
    index += 1
    if root.v != '#':
        root.left = build()
        root.right = build()
    return root


while True:
    try:
        n = int(input())
        if n == 0:
            break
        s = input().split()
        index = 0
        build()
        print('T' if index == n else 'F')
    except:
        print('F')
```

前中建树（转后）

```
def build_tree(preorder, inorder):
    if not preorder:
        return ''
```

```python
    root = preorder[0]
    root_index = inorder.index(root)

    left_preorder = preorder[1:1 + root_index]
    right_preorder = preorder[1 + root_index:]

    left_inorder = inorder[:root_index]
    right_inorder = inorder[root_index + 1:]

    left_tree = build_tree(left_preorder, left_inorder)
    right_tree = build_tree(right_preorder, right_inorder)

    return left_tree + right_tree + root
```

中后建树

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

def build_tree(inorder, postorder):
    if inorder:
        root = Node(postorder.pop())
        root_index = inorder.index(root.data)
        root.right = build_tree(inorder[root_index+1:], postorder)
        root.left = build_tree(inorder[:root_index], postorder)
        return root
```

二叉堆

```python
class tree_node:
    def __init__(self):
        self.heap = []

    def parent(self, i):
        return (i - 1) // 2

    def left_child(self, i):
        return 2 * i + 1

    def right_child(self, i):
```

```
        return 2 * i + 2

    def swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def insert(self, item):
        self.heap.append(item)
        self.heapify_up(len(self.heap) - 1)

    def delete(self):
        if len(self.heap) == 0:
            raise IndexError("Heap is empty")

        self.swap(0, len(self.heap) - 1)
        min_value = self.heap.pop()
        self.heapify_down(0)
        return min_value

    def heapify_up(self, i):
        while i > 0 and self.heap[i] < self.heap[self.parent(i)]:
            self.swap(i, self.parent(i))
            i = self.parent(i)

    def heapify_down(self, i):
        min_index = i
        left = self.left_child(i)
        right = self.right_child(i)

        if left < len(self.heap) and self.heap[left] <
self.heap[min_index]:
            min_index = left

        if right < len(self.heap) and self.heap[right] <
self.heap[min_index]:
            min_index = right

        if i != min_index:
            self.swap(i, min_index)
            self.heapify_down(min_index)
n = int(input())
lst = tree_node()
for _ in range(n):
    s = input()
    if s[0] == '1':
```

```
        lst.insert(int(s[2:]))
    if s[0] == '2':
        print(lst.delete())
```

搜索树

二叉搜索树

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def insert(node, value):
    if node is None:
        return TreeNode(value)
    if value < node.value:
        node.left = insert(node.left, value)
    elif value > node.value:
        node.right = insert(node.right, value)
    return node
```

AVL树

```
from collections import deque    # 在遍历的时候使用而已，其实并不需要
class TreeNode():
    def __init__(self):       # 这是没有设置父连接的版本
        self.data=0
        self.left=None        # 左孩子
        self.right=None       # 右孩子
        self.height=0         # 节点高度（这里设置所有节点的初始高度为0，后续会根据情况调整，根节点的高度会随之增加）

class BTree():
    def __init__(self):
        self.root=None
    def __Max(self,h1,h2):        # 比较左右孩子的高度，返回最大的那个
        if h1>h2:
            return h1
```

```
        elif h1<=h2:
            return h2

    def __LL(self,r):#左左情况，向右旋转      r是第一个高度差不满足的节点
        node=r.left
        r.left=node.right
        node.right=r

 r.height=self.__Max(self.getHeight(r.right),self.getHeight(r.left))+1

 node.height=self.__Max(self.getHeight(node.right),self.getHeight(node.l
eft))+1
        return node           # 返回的是最开始r的左孩子，也是现在子树中的根节点
    def __RR(self,r):#右右，左旋           r是第一个高度差不满足条件的节点
        node = r.right
        r.right = node.left
        node.left = r
        r.height = self.__Max(self.getHeight(r.right),
self.getHeight(r.left)) + 1
        node.height = self.__Max(self.getHeight(node.right),
self.getHeight(node.left)) + 1
        return node           # 返回的是最开始r的右孩子，也是现在子树中的根节点
    def __LR(self,r):#左右，先左旋再右旋       r是第一个高度差不满足条件的节点
        r.left=self.__RR(r.left)          # 对r的左孩子进行左旋
        return self.__LL(r)               # 对r进行右旋，随即返回
    def __RL(self,r):#右左，先右旋再左旋       r是第一个高度差不满足条件的节点
        r.right=self.__LL(r.right)        # 对r的右孩子进行右旋
        return self.__RR(r)               # 对r进行左旋，随即返回
    def __insert(self,data,r):
        if r==None:               # 假入没有根节点，就新创建一个二叉树，插入值作
为根节点的值
            node=TreeNode()
            node.data=data
            return node
        elif data==r.data:        # 假设插入值和根节点相同，那么就直接返回根节点
            return r
        elif data<r.data:         # 假设插入值比根节点的值要小，使用递归迭代，
            r.left=self.__insert(data,r.left)
            if self.getHeight(r.left)-self.getHeight(r.right)>=2:    # 如
果左孩子的高度 比 右孩子的高度 大于等于2；简单的说其实是在判定是在左孩子的子树进行插入
                if data<r.left.data:    # 插入数值比左孩子要小（即在左子树中插
入），使用右旋
                    r=self.__LL(r)
                else:                   # 插入数值比右孩子要大（即在右子树中插
入），使用左旋-右旋
```

```
                    r=self.__LR(r)
        else:
            r.right=self.__insert(data,r.right)
            if self.getHeight(r.right)-self.getHeight(r.left)>=2:    # 如果右孩子的高度 比 左孩子的高度 大于等于2；简单的说其实是在判定是在右孩子的子树进行插入
                if data>r.right.data:    # 插入数值比右孩子要大（即在右子树中插入），使用左旋
                    r=self.__RR(r)
                else:                    # 插入数值比左孩子要小（即在左子树中插入），使用右旋-左旋
                    r=self.__RL(r)

 r.height=self.__Max(self.getHeight(r.left),self.getHeight(r.right))+1
   # 修正树的根节点的深度
        return r

    # 删除data节点
    def __delete(self,data,r):
        if r==None:                    # 假如节点为空，就返回
            print("don't have %d"%data)
            return r
        elif r.data==data:             # 当节点的值和删除值相同时
            if r.left==None:           #如果被删除的节点只有右子树，直接将右子树赋值到此节点
                return r.right
            elif r.right==None:        #如果被删除的节点只有左子树，直接将左子树赋值到此节点
                return r.left
            else:#如果同时有左右子树
                if self.getHeight(r.left)>self.getHeight(r.right):  #左子树高度大于右子树
                    # 找到左子树的最右节点（最大值） 返回节点值 并删除该节点
                    node=r.left
                    while(node.right!=None):
                        node=node.right
                    r=self.__delete(node.data,r)       # 调用自身删除node
                    r.data=node.data
                    return r
                else:                              # 右子树高度大于左子树
                    node=r.right
                    # 找到右子树的最小节点（最小值） 返回节点值 并删除该节点
                    while node.left!=None:
                        node=node.left
                    r=self.__delete(node.data,r)       # 调用自身删除node
                    r.data=node.data
```

```
                    return r
        elif data<r.data:                    # 当删除值小于根节点的值时
            r.left=self.__delete(data,r.left)            # 在左子树中删除,使用递归删除
            if self.getHeight(r.right)-self.getHeight(r.left)>=2:    # 删除后，如果右子树高度与左子树高度相差超过1
                if self.getHeight(r.right.left)>self.getHeight(r.right.right):
                    r=self.__RL(r)            # 第一个错误点在右孩子的左子树中，使用右旋-左旋
                else:
                    r=self.__RR(r)            # 第一个错误点在右孩子的右子树中，使用左旋
        elif data>r.data:                   # 当删除值大于根节点的值时
            r.right=self.__delete(data,r.right)          # 右子树中删除
            if self.getHeight(r.left)-self.getHeight(r.right)>=2: # 左子树与右子树高度相差超过1
                if self.getHeight(r.left.right)>self.getHeight(r.left.left):
                    r=self.__LR(r)            # 第一个错误点在左孩子的右子树中，使用左旋-右旋
                else:
                    r=self.__LL(r)            # 第一个错误点在左孩子的左子树中，使用右旋

 r.height=self.__Max(self.getHeight(r.left),self.getHeight(r.right))+1
         # 根节点的高度为 左右孩子节点最大的那个+1
        return r

    def Insert(self,data):                  # 插入操作
        self.root=self.__insert(data,self.root)          #  把新插入的节点从根节点开始比较
        return self.root            # 返回根节点
    def Delete(self,data):                  # 删除操作
        self.root=self.__delete(data,self.root)          # 从根节点开始寻找，找到要删除的节点位置
        return self.root
```

哈夫曼编码

```
import heapq

class Node:
```

```python
    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.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:
            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
```

```python
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.char:
            decoded += node.char
            node = root
    return decoded
```

并查集

普通并查集

```python
def find_set(x):    #有路径压缩优化的查询
    if x != s[x]:                  # 不等于自己的集
        s[x] = find_set(s[x])     # 把集改成根节点的集
    return s[x]

def merge_set(x,y):  #合并
    x = find_set(x)
    y = find_set(y)
    if x != y: s[x] = s[y]

n,m = map(int,input().split())
s = list (range(10**6))  # 大小看题目要求，最大规模10**6，或者n+10也可以
for i in range(m):
    op,x,y = map(int,input().split())
    if op == 1:
        merge_set(x,y)
    else:
        if find_set(x) == find_set(y):    # 两个人的组织相同，则是朋友
            print("YES")
        else:
            print("NO")
```

带权并查集

```
def find(_x):
    if _x != pre[_x]:
        t = find(pre[_x])
        h[_x] = (h[pre[_x]] + h[_x]) % multi_num
        pre[_x] = t
    return pre[_x]


def merge(_a, _b, sign):
    _fa, _fb = find(_a), find(_b)
    pre[_fa] = _fb
    h[_fa] = h[_b] - h[_a] + sign


def same(_a, _b):
    return (h[a] - h[b]) % multi_num == 0 and find(_a) == find(_b)


T = int(input())
multi_num = 2
for _ in range(T):
    n, k = map(int, input().split())
    pre = [i for i in range(n)]
    h = [0 for _ in range(n)]
    for __ in range(k):
        op, a, b = input().split()
        a, b = int(a) - 1, int(b) - 1
        fa, fb = find(a), find(b)
        if op == 'A':
            if find(a) != find(b):
                print('Not sure yet.')
            elif same(a, b):
                print('In the same gang.')
            else:
                print('In different gangs.')
        elif op == 'D':
            merge(a, b, 1)
```

中序转后序

```
def infix_to_postfix(expression):
```

```python
    precedence = {'+':1, '-':1, '*':2, '/':2}
    stack = []
    postfix = []
    number = ''

    for char in expression:
        if char.isnumeric() or char == '.':
            number += char
        else:
            if number:
                num = float(number)
                postfix.append(int(num) if num.is_integer() else num)
                number = ''
            if char in '+-*/':
                while stack and stack[-1] in '+-*/' and precedence[char]
<= precedence[stack[-1]]:
                    postfix.append(stack.pop())
                stack.append(char)
            elif char == '(':
                stack.append(char)
            elif char == ')':
                while stack and stack[-1] != '(':
                    postfix.append(stack.pop())
                stack.pop()

    if number:
        num = float(number)
        postfix.append(int(num) if num.is_integer() else num)

    while stack:
        postfix.append(stack.pop())

    return ' '.join(str(x) for x in postfix)

n = int(input())
for _ in range(n):
    expression = input()
    print(infix_to_postfix(expression))
```

图算法

BFS

```
q = deque()
q.append((0, 0, 0))
v = set()
steps = -1
while q:
    x0, y0, h = q.popleft()
    if pct[x0][y0] == 1:
        steps = h
        break
    v.add((x0, y0))
    for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
        x1 = x0 + dx
        y1 = y0 + dy
        if 0 <= x1 < m and 0 <= y1 < n and pct[x1][y1] != 2 and (x1, y1)
not in v:
            q.append((x1, y1, h + 1))
```

Dijkstra

```
def dijkstra(graph: List[List[int]], n: int, k: int):
  # 稀疏图用邻接表graph, 其中graph[node]存储以(nb, w)形式存储邻居和边权
  expanded = [False for _ in range(n)]
  curDist = [float('inf') for _ in range(n)]
  curDist[k] = 0

  h = [(0, k)]
  while h:
    # 1. 找到没有扩展过的点中到起点距离最短的点node
    node = heappop(h)
    if not expanded[node]: # 确保不重复扩展
      # 2. 扩展
      for nb, w in graph[node]:
        # 扩展意味着搜索树中nb(或暂时成为, 取决于搜索树类型)node的子节点
        newDist = curDist[node] + w
        if newDist < curDist[nb]: # 剪枝
          curDist[nb] = newDist
          heappush(h, (newDist, nb))
      # 3. 扩展完标记该点
      expanded[node] = True
```

道路

```python
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)
```

记录路径

```python
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
```

```python
    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)
```

Floyd

```
def floyd_warshall(graph):
    n = len(graph)
    dist = [[float('inf')] * n for _ in range(n)]

    for i in range(n):
        for j in range(n):
            if i == j:
                dist[i][j] = 0
            elif j in graph[i]:
                dist[i][j] = graph[i][j]

    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist
```

最小生成树

Prim

```
def prim(graph: List[List[int]], n: int):
  # 稠密图用邻接矩阵graph, 其中存边权, 无边存无穷大
  curDist = [float('inf') for _ in range(n)] # 点到当前树的最小距离, 是边权
  inMST = [False for _ in range(n)] # 标记是否已加入到MST中
```

```
  totalWeight = 0

  for _ in range(n): # 每次加一个点一条边到树中(第一次只加点不加边)
    # 1. 通过枚举点找到连接树和树外一点的最短边
    minNode = None
    for node in range(n):
      if not inMST[node] and (minNode is None
          \ or curDist[node] < curDist[minNode]):
        minNode = nodei

    # 2. 把最短边及其连接的树外点加入到MST中(第一次循环只加点不加边)
    if i != 0:  # 当然也可将起点的 curDist 初始化为 0, 则此处无需判断
      totalWeight += curDist[minNode]
      # 如果这条最短边为inf, 就代表该树外点与树中任一点都不连通, 即原图是不连通的
    inMST[minNode] = True

    # 3. 更新树外节点到树的最小距离
    for nb in graph[minNode]:
      curDist[nb] = min(curDist[nb], graph[minNode][nb])

  return totalWeight
```

```
from heapq import *

while True:
    n = int(input())
    if n == 0:
        break
    trucks = [input() for _ in range(n)]
    trucks.sort()

    sd = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        for j in range(i + 1, n):
            sd[i][j] = sd[j][i] = sum(a != b for a, b in zip(trucks[i],
trucks[j]))

    # Prim
    v = [False for _ in range(n)]
    dis = [float('inf') for _ in range(n)]
    dis[0] = 0
    q = [(0, 0)]
    total_weight = 0
    while q:
```

```python
        weight, node = heappop(q)
        if v[node]:
            continue
        v[node] = True
        total_weight += weight
        for nb in range(n):
            if nb != node and not v[nb] and dis[nb] > sd[nb][node]:
                dis[nb] = sd[nb][node]
                heappush(q, (dis[nb], nb))

    print(f'The highest possible quality is 1/{total_weight}.')
```

拓扑排序

DFS

```python
    def dfs(num):
        v[num] = 1
        for neighbour in g[num]:
            if v[neighbour] == 0 and dfs(neighbour):
                return True
            if v[neighbour] == 1:
                return True
        v[num] = 2
        ans.append(num) # 最后要反转
        return False
```

Kahn

```python
def topo_sort(graph):
    in_degree = {u:0 for u in graph}
    for u in graph:
        for v in graph[u]:
            in_degree[v] += 1
    q = deque([u for u in in_degree if in_degree[u] == 0])
    topo_order = [];flag = True
    while q:
        if len(q) > 1:
            flag = False#topo_sort不唯一确定
        u = q.popleft()
        topo_order.append(u)
        for v in graph[u]:
```

```
            in_degree[v] -= 1
            if in_degree[v] == 0:
                q.append(v)
    if len(topo_order) != len(graph): return 0
    return topo_order if flag else None
```

强连通分量

```
def dfs1(graph, node, visited, stack):
    visited[node] = True
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs1(graph, neighbor, visited, stack)
    stack.append(node)

def dfs2(graph, node, visited, component):
    visited[node] = True
    component.append(node)
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs2(graph, neighbor, visited, component)

def kosaraju(graph):
    # Step 1: Perform first DFS to get finishing times
    stack = []
    visited = [False] * len(graph)
    for node in range(len(graph)):
        if not visited[node]:
            dfs1(graph, node, visited, stack)

    # Step 2: Transpose the graph
    transposed_graph = [[] for _ in range(len(graph))]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)

    # Step 3: Perform second DFS on the transposed graph to find SCCs
    visited = [False] * len(graph)
    sccs = []
    while stack:
        node = stack.pop()
        if not visited[node]:
            scc = []
            dfs2(transposed_graph, node, visited, scc)
```

```
            sccs.append(scc)
    return sccs

# Example
graph = [[1], [2, 4], [3, 5], [0, 6], [5], [4], [7], [5, 6]]
sccs = kosaraju(graph)
print("Strongly Connected Components:")
for scc in sccs:
    print(scc)

"""
Strongly Connected Components:
[0, 3, 2, 1]
[6, 7]
[5, 4]

"""
```

网络最大流

```
import numpy as np


class Node:
    def __init__(self, name, arc_dict):
        self.name = name
        self.arc_dict = arc_dict


def create_node(name, next_list, flow_list):
    arc_dict = {}
    for i in range(len(next_list)):
        arc_dict[next_list[i]] = flow_list[i]
    return Node(name, arc_dict)


def create_level_graph(s, e, node_list, name_index_dict):
    level_graph = np.zeros((len(node_list), len(node_list))).tolist()
    cur_layer = [s]
    all_node = set()
    all_node.add(s)
    next_layer = set()
    while len(cur_layer) > 0:
        for node_name in cur_layer:
```

```
            node = node_list[name_index_dict[node_name]]
            for key in node.arc_dict.keys():
                if key not in all_node and (node.arc_dict[key] is None
or node.arc_dict[key] > 0):
                    level_graph[name_index_dict[node_name]]
[name_index_dict[key]] = node.arc_dict[key]
                    next_layer.add(key)
                    all_node.add(key)
        cur_layer = list(next_layer)
        next_layer = set()
    return level_graph if e in all_node else None


def Dinic_Solve(s, e, node_list, name_index_dict):
    routes = []
    s_index = name_index_dict[s]
    e_index = name_index_dict[e]
    level_graph = create_level_graph(s, e, node_list, name_index_dict)
    while level_graph is not None:
        res_list = []
        while True:
            res = dfs(e_index, [s_index], None, level_graph)
            if res is None:
                break
            # 更新 level graph
            route, flow = res
            for i in range(len(route) - 1):
                if level_graph[route[i]][route[i + 1]] is not None:
                    level_graph[route[i]][route[i + 1]] -= flow
            # 追加记录增广路径
            res_list.append(res)
            routes.append([[node_list[n].name for n in res[0]], res[1]])
        # 更新残存网络
        for res in res_list:
            update(res, node_list)
        # 重新构造 level graph
        level_graph = create_level_graph(s, e, node_list,
name_index_dict)
    return routes


def update(res, node_list):
    route, flow = res
    for i in range(len(route) - 1):
        n1 = node_list[route[i]]
```

```python
        n2 = node_list[route[i + 1]]
        # 正向更新 n1 -> n2 剩余流量减少
        if n2.name in n1.arc_dict.keys() and n1.arc_dict[n2.name] is not None:
            n1.arc_dict[n2.name] = n1.arc_dict[n2.name] - flow
        # 反向更新 n2 -> n1 剩余流量增加
        if n1.name in n2.arc_dict.keys() and n2.arc_dict[n1.name] is not None:
            n2.arc_dict[n1.name] = n2.arc_dict[n1.name] + flow


def dfs(e_index, cur_route, last_flow, level_graph):
    if cur_route[-1] == e_index:
        return cur_route, last_flow
    for next_node in range(len(level_graph)):
        if next_node not in cur_route:
            if level_graph[cur_route[-1]][next_node] is None or level_graph[cur_route[-1]][next_node] > 0:
                flow = min_flow(level_graph[cur_route[-1]][next_node], last_flow)
                cur_route.append(next_node)
                res = dfs(e_index, cur_route, flow, level_graph)
                if res is not None:
                    return res
                cur_route.pop(-1)


def min_flow(f1, f2):
    '''
    求两个流量的较小者
    '''
    if f1 is None:
        return f2
    elif f2 is None:
        return f1
    else:
        return min(f1, f2)


if __name__ == '__main__':
    # 格式：[节点名，后继节点的名称，当前节点到各个后继的流量] （None 代表流量无穷大）
    graph = [
        ["S", ["1", "2", "3"], [None, None, None]],
        ["1", ["4"], [1]],
```

```
        ["2", ["4", "6"], [1, 1]],
        ["3", ["5"], [1]],
        ["4", ["1", "2", "E"], [0, 0, 1]],
        ["5", ["3", "E"], [0, 1]],
        ["6", ["2", "E"], [0, 1]],
        ["E", [], []]
    ]
    name_index_dict = dict()
    node_list = []
    for i in range(len(graph)):
        node_list.append(create_node(graph[i][0], graph[i][1], graph[i]
[2]))
        name_index_dict[graph[i][0]] = i

    # 调用算法求解最大流
    routes = Dinic_Solve("S", "E", node_list, name_index_dict)
    for i, (route, flow) in enumerate(routes):
        print(f"Route-{i + 1}: {route} , flow: {flow}")
```

波兰表达式

```
s = input().split()
def cal():
    cur = s.pop(0)
    if cur in "+-*/":
        return str(eval(cal() + cur + cal()))
    else:
        return cur
print("%.6f" % float(cal()))
```