```python
import sys
import heapq
from collections import deque, defaultdict

def main():
    lines = [line.strip() for line in sys.stdin if line.strip()]
    if not lines:
        return []
    for line in lines[0:]:
        pass
    return

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

易错点

注意可能的输入问题如可能有个输入是空的

注意数据类型的转化（比如放到列表的序号的得是整数）

一个条件语句内前面的改动可能影响后面！

注意str不能直接用于拼接列表

常用函数与数据结构

sorted函数

基本语法

```python
sorted(iterable, *, key=None, reverse=False)
```

- `iterable`: 要排序的可迭代对象（如列表、元组、字符串等）
- `key`: 排序依据的函数（可选）
- `reverse`: 是否逆序排序（默认为False，即升序）

2. 对字符串排序

```python
text = "python"
sorted_text = sorted(text)
print(sorted_text)  # 输出: ['h', 'n', 'o', 'p', 't', 'y']
# 注意：返回的是字符列表，可以使用join()合并
print(''.join(sorted_text))  # 输出: hnopty
```

使用key参数

`key`参数指定一个函数，用于从每个元素中提取比较键。

1. 按长度排序字符串

```
words = ["banana", "pie", "apple", "orange"]
sorted_words = sorted(words, key=len)
print(sorted_words)  # 输出: ['pie', 'apple', 'banana', 'orange']
```

2. 按第二个元素排序

```
pairs = [(1, 'one'), (3, 'three'), (2, 'two')]
sorted_pairs = sorted(pairs, key=lambda x: x[1])
print(sorted_pairs)  # 输出: [(1, 'one'), (3, 'three'), (2, 'two')]
```

3. 使用attrgetter排序对象

```
from operator import attrgetter

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

people = [Person("Alice", 25), Person("Bob", 20), Person("Charlie", 30)]
sorted_people = sorted(people, key=attrgetter('age'))
print([p.name for p in sorted_people])  # 输出: ['Bob', 'Alice', 'Charlie']
```

使用reverse参数

```
numbers = [3, 1, 4, 1, 5, 9, 2]
sorted_numbers_desc = sorted(numbers, reverse=True)
print(sorted_numbers_desc)  # 输出: [9, 5, 4, 3, 2, 1, 1]
```

与list.sort()的区别

- `sorted()`返回一个新列表，原列表不变
- `list.sort()`是列表方法，原地修改列表，返回None

```
lst = [3, 1, 2]
new_lst = sorted(lst)  # lst不变，new_lst是排序后的新列表
lst.sort()  # lst被原地修改
```

多级排序

```
students = [
    {'name': 'Alice', 'grade': 'A', 'age': 20},
    {'name': 'Bob', 'grade': 'B', 'age': 21},
    {'name': 'Charlie', 'grade': 'A', 'age': 19}
]

# 先按grade升序，再按age升序
sorted_students = sorted(students, key=lambda x: (x['grade'], x['age']))
```

heapq

`heapq`是Python的一个内置模块，提供了堆队列算法的实现，也称为优先队列算法。堆是一种特殊的二叉树结构，满足父节点的值总是小于或等于其子节点的值（最小堆）。

创建堆

Python中的堆通常用列表表示，`heapq`模块提供了将列表转换为堆的函数：

```
import heapq

# 创建一个列表
data = [3, 1, 4, 1, 5, 9, 2, 6]

# 使用heapify将列表转换为堆（原地修改）
heapq.heapify(data)
print(data)  # 输出可能是 [1, 1, 2, 3, 5, 9, 4, 6]
```

添加和弹出元素

```
# 添加元素到堆
heapq.heappush(data, 0)
print(data)  # 0会被放在堆顶

# 弹出最小元素
smallest = heapq.heappop(data)
print(smallest)  # 输出0
print(data)      # 堆结构自动调整
```

查看最小元素

```
# 查看最小元素而不弹出
print(data[0])  # 堆的第一个元素总是最小的
```

合并多个堆

```
heap1 = [1, 3, 5]
heap2 = [2, 4, 6]
merged = list(heapq.merge(heap1, heap2))
print(merged)  # 输出 [1, 2, 3, 4, 5, 6]
```

获取前N个最大/最小元素

```
nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]

# 获取3个最大的元素
print(heapq.nlargest(3, nums))  # [42, 37, 23]

# 获取3个最小的元素
print(heapq.nsmallest(3, nums))  # [-4, 1, 2]
```

带键函数的堆操作

```
portfolio = [
    {'name': 'IBM', 'shares': 100, 'price': 91.1},
    {'name': 'AAPL', 'shares': 50, 'price': 543.22},
    {'name': 'FB', 'shares': 200, 'price': 21.09},
    {'name': 'HPQ', 'shares': 35, 'price': 31.75},
    {'name': 'YHOO', 'shares': 45, 'price': 16.35},
    {'name': 'ACME', 'shares': 75, 'price': 115.65}
]

# 获取价格最低的2个
cheap = heapq.nsmallest(2, portfolio, key=lambda s: s['price'])
print(cheap)
# 输出: [{'name': 'YHOO', 'shares': 45, 'price': 16.35},
#        {'name': 'FB', 'shares': 200, 'price': 21.09}]
```

自定义堆元素比较

如果需要自定义比较逻辑，可以创建一个包装类：

```
class PriorityItem:
    def __init__(self, priority, item):
        self.priority = priority
        self.item = item

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

heap = []
heapq.heappush(heap, PriorityItem(3, "C"))
heapq.heappush(heap, PriorityItem(1, "A"))
heapq.heappush(heap, PriorityItem(2, "B"))
```

```
while heap:
    item = heapq.heappop(heap)
    print(item.item)  # 输出 A, B, C
```

注意事项

1. `heapq`模块实现的是最小堆，如果需要最大堆，可以将元素取负数存储
2. 堆操作的时间复杂度：
    - `heappush`和`heappop`: O(log n)
    - `heapify`: O(n)
    - `nlargest`和`nsmallest`: O(n log k)，k是要返回的元素数量
3. 堆结构不保证列表完全有序，只保证堆顶元素是最小的

`heapq`模块非常适合需要频繁获取最小或最大元素的场景，如任务调度、Top K问题等。

deque

deque（双端队列）是 Python 中`collections`模块提供的一个数据结构，它允许在队列的两端高效地添加和删除元素。以下是 deque 的基本使用方法：

基本操作

```
from collections import deque

# 创建一个 deque
d = deque()  # 空 deque
d = deque([1, 2, 3])  # 用可迭代对象初始化

# 添加元素
d.append(4)      # 在右端添加元素 -> deque([1, 2, 3, 4])
d.appendleft(0)  # 在左端添加元素 -> deque([0, 1, 2, 3, 4])

# 删除元素
d.pop()          # 移除并返回右端元素 -> 4
d.popleft()      # 移除并返回左端元素 -> 0
```

其他常用方法

```
# 扩展 deque
d.extend([4, 5])        # 在右端扩展 -> deque([1, 2, 3, 4, 5])
d.extendleft([0, -1])  # 在左端扩展（注意顺序） -> deque([-1, 0, 1, 2, 3, 4,
5])

# 旋转元素
d.rotate(1)   # 向右旋转1位 -> deque([5, -1, 0, 1, 2, 3, 4])
d.rotate(-1)  # 向左旋转1位 -> deque([-1, 0, 1, 2, 3, 4, 5])

# 限制 deque 大小
d = deque(maxlen=3)  # 创建固定长度的 deque
d.extend([1, 2, 3]) # deque([1, 2, 3], maxlen=3)
d.append(4)          # 自动移除最左端元素 -> deque([2, 3, 4], maxlen=3)
```

defaultdict

基本使用

```
from collections import defaultdict

# 创建一个默认值为list的defaultdict
dd = defaultdict(list)

# 添加元素
dd['fruits'].append('apple')
dd['fruits'].append('banana')
dd['vegetables'].append('carrot')
#添加多个元素
dd['fruits'].extend(['apple', 'banana'])

print(dd)
# 输出: defaultdict(<class 'list'>, {'fruits': ['apple', 'banana'],
'vegetables': ['carrot']})
```

str

list

set

线性表

递归与动规，KMP

总结

类型	模板名	说明
背包 DP	0-1 背包	每个物品只能选一次
背包 DP	完全背包	每个物品可选无限次
计数型 DP	硬币组合	求凑成目标值的方案数（完全背包）
最长序列	LIS	最长上升子序列
搜索 + 记忆化	dfs + `@lru_cache`	高效递归状态转移（树形/状态压缩）
区间型 DP	区间DP	区间最优分割（石子合并等）
字符串算法	KMP	快速字符串匹配，O(n + m)

1. 0-1 背包（每件物品只能选一次）

```python
def knapsack_01(W, w, v):
    # W: 总容量，w: 重量数组，v: 价值数组
    n = len(w)
    dp = [0] * (W + 1)
    for i in range(n):
        for j in range(W, w[i] - 1, -1):  # 从后往前，防止重复选择
            dp[j] = max(dp[j], dp[j - w[i]] + v[i])
    return dp[W]
```

2. 完全背包（每件物品可选无限次）

```python
def knapsack_complete(W, w, v):
    # W: 总容量，w: 重量数组，v: 价值数组
    n = len(w)
    dp = [0] * (W + 1)
    for i in range(n):
        for j in range(w[i], W + 1):  # 正序遍历，允许重复选
            dp[j] = max(dp[j], dp[j - w[i]] + v[i])
    return dp[W]
```

3. 完全背包计数型变种：凑硬币的方案数

```python
def coin_change_count(coins, target):
    # coins: 硬币面值数组，target: 目标金额
    dp = [0] * (target + 1)
    dp[0] = 1  # 凑出0元只有1种方法：什么都不选
    for coin in coins:
        for i in range(coin, target + 1):  # 顺序遍历：可重复使用
            dp[i] += dp[i - coin]
    return dp[target]
```

4. 最长上升子序列（LIS）

```
def lengthOfLIS(nums):
    # 返回最长上升子序列的长度
    n = len(nums)
    dp = [1] * n  # dp[i]: 以 nums[i] 结尾的 LIS 长度
    for i in range(n):
        for j in range(i):
            if nums[j] < nums[i]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)
```

5. 记忆化搜索模板（递归 + 缓存）

```
from functools import lru_cache

def dfs_with_memo(n):
    # 示例：计算斐波那契数
    @lru_cache(None)  # 自动缓存函数结果
    def dfs(k):
        if k <= 1:
            return k
        return dfs(k - 1) + dfs(k - 2)

    return dfs(n)
```

> 📌 一般形式：用 `@lru_cache` 包装递归函数，可用于树形 DP、博弈 DP、状态压缩等情景。

6. 区间 DP 模板（区间切割问题，典型如戳气球、石子合并）

```
def interval_dp(nums):
    n = len(nums)
    dp = [[0] * n for _ in range(n)]
    for l in range(2, n+1):  # 区间长度
        for i in range(n - l + 1):
            j = i + l - 1
            for k in range(i+1, j):
                # 样例递推：dp[i][j] = max/min(dp[i][k] + dp[k][j] +
cost[i][j])
                dp[i][j] = max(dp[i][j], dp[i][k] + dp[k][j] +
nums[i]*nums[k]*nums[j])
    return dp[0][n-1]
```

> 📌 适用于问题形如："将区间 `[i,j]` 拆成若干部分，求最小/最大代价"。

7. KMP 字符串匹配算法

```python
def kmp(s, p):
    # s: 主串，p: 模式串
    n, m = len(s), len(p)
    lps = [0] * m  # 最长前后缀数组
    j = 0
    for i in range(1, m):
        while j > 0 and p[i] != p[j]:
            j = lps[j - 1]
        if p[i] == p[j]:
            j += 1
        lps[i] = j

    res = []
    j = 0
    for i in range(n):
        while j > 0 and s[i] != p[j]:
            j = lps[j - 1]
        if s[i] == p[j]:
            j += 1
        if j == m:
            res.append(i - m + 1)  # 匹配起点
            j = lps[j - 1]
    return res  # 返回所有匹配位置
```

> 📌 时间复杂度 `O(n + m)`，比暴力 `O(n * m)` 更快。

排序与查找

二分

见1776

树及算法

二叉树相关结构与算法实现合集，包括：

1. 基础类定义（树节点）
2. 遍历（前中后层序）
3. 解析树构建与求值
4. 二叉堆实现（最小堆）
5. 二叉搜索树（BST）
6. AVL 树（含旋转）

1. 树的基本定义

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

2. 二叉树的遍历

```
def preorder(node):
    if node:
        print(node.val, end=" ")
        preorder(node.left)
        preorder(node.right)

def inorder(node):
    if node:
        inorder(node.left)
        print(node.val, end=" ")
        inorder(node.right)

def postorder(node):
    if node:
        postorder(node.left)
        postorder(node.right)
        print(node.val, end=" ")

from collections import deque
def level_order(node):
    if not node:
        return
    queue = deque([node])
    while queue:
        curr = queue.popleft()
        print(curr.val, end=" ")
        if curr.left: queue.append(curr.left)
        if curr.right: queue.append(curr.right)
```

3. 解析树（Parse Tree）构建与求值

```
import operator

def build_parse_tree(expr):
    tokens = expr.split()
    stack = []
    root = BinaryTreeNode('')
    current = root
    for token in tokens:
        if token == '(':
            current.left = BinaryTreeNode('')
            stack.append(current)
```

```
            current = current.left
        elif token in '+-*/':
            current.val = token
            current.right = BinaryTreeNode('')
            stack.append(current)
            current = current.right
        elif token == ')':
            current = stack.pop()
        else:  # 数字
            current.val = int(token)
            current = stack.pop()
    return root

def evaluate_parse_tree(node):
    ops = {'+': operator.add, '-': operator.sub,
           '*': operator.mul, '/': operator.truediv}
    if not node.left and not node.right:
        return node.val
    left_val = evaluate_parse_tree(node.left)
    right_val = evaluate_parse_tree(node.right)
    return ops[node.val](left_val, right_val)
```

4. 二叉堆（最小堆）

```
class MinHeap:
    def __init__(self):
        self.heap = [0]  # 占位

    def insert(self, val):
        self.heap.append(val)
        self._up(len(self.heap) - 1)

    def _up(self, idx):
        while idx // 2 > 0:
            if self.heap[idx] < self.heap[idx // 2]:
                self.heap[idx], self.heap[idx // 2] = self.heap[idx // 2],
self.heap[idx]
            idx //= 2

    def delete_min(self):
        if len(self.heap) == 1:
            return None
        min_val = self.heap[1]
        self.heap[1] = self.heap[-1]
        self.heap.pop()
        self._down(1)
        return min_val

    def _down(self, idx):
        while idx * 2 < len(self.heap):
            mc = self._min_child(idx)
```

```
            if self.heap[idx] > self.heap[mc]:
                self.heap[idx], self.heap[mc] = self.heap[mc],
self.heap[idx]
            idx = mc

    def _min_child(self, idx):
        if idx * 2 + 1 >= len(self.heap):
            return idx * 2
        if self.heap[idx * 2] < self.heap[idx * 2 + 1]:
            return idx * 2
        return idx * 2 + 1
```

5. 二叉搜索树（BST）

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

class BST:
    def insert(self, root, val):
        if not root:
            return BSTNode(val)
        if val < root.val:
            root.left = self.insert(root.left, val)
        else:
            root.right = self.insert(root.right, val)
        return root

    def search(self, root, val):
        if not root or root.val == val:
            return root
        if val < root.val:
            return self.search(root.left, val)
        return self.search(root.right, val)

    def delete(self, root, val):
        if not root:
            return root
        if val < root.val:
            root.left = self.delete(root.left, val)
        elif val > root.val:
            root.right = self.delete(root.right, val)
        else:
            if not root.left:
                return root.right
            if not root.right:
                return root.left
            # 找最小值替代
            min_larger_node = self._min_value(root.right)
            root.val = min_larger_node.val
```

```
            root.right = self.delete(root.right, min_larger_node.val)
        return root

    def _min_value(self, node):
        while node.left:
            node = node.left
        return node
```

6. AVL 树（自动平衡）

```
class AVLNode:
    def __init__(self, val):
        self.val = val
        self.left = self.right = None
        self.height = 1

class AVLTree:
    def height(self, node):
        return node.height if node else 0

    def get_balance(self, node):
        return self.height(node.left) - self.height(node.right) if node
else 0

    def rotate_right(self, y):
        x = y.left
        T = x.right
        x.right = y
        y.left = T
        y.height = 1 + max(self.height(y.left), self.height(y.right))
        x.height = 1 + max(self.height(x.left), self.height(x.right))
        return x

    def rotate_left(self, x):
        y = x.right
        T = y.left
        y.left = x
        x.right = T
        x.height = 1 + max(self.height(x.left), self.height(x.right))
        y.height = 1 + max(self.height(y.left), self.height(y.right))
        return y

    def insert(self, node, key):
        if not node:
            return AVLNode(key)
        if key < node.val:
            node.left = self.insert(node.left, key)
        else:
            node.right = self.insert(node.right, key)
```

```
        node.height = 1 + max(self.height(node.left),
self.height(node.right))
        balance = self.get_balance(node)

        if balance > 1 and key < node.left.val:
            return self.rotate_right(node)
        if balance < -1 and key > node.right.val:
            return self.rotate_left(node)
        if balance > 1 and key > node.left.val:
            node.left = self.rotate_left(node.left)
            return self.rotate_right(node)
        if balance < -1 and key < node.right.val:
            node.right = self.rotate_right(node.right)
            return self.rotate_left(node)

        return node
```

图与算法

bfs

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])

    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=' ')
            visited.add(node)
            queue.extend(neighbor for neighbor in graph[node] if neighbor
not in visited)
```

尤其注意边界的判定!

dfs

递归版

```
def dfs_recursive(graph, node, visited=None):
    if visited is None:
        visited = set()
    if node not in visited:
        print(node, end=' ')
        visited.add(node)
        for neighbor in graph[node]:
            dfs_recursive(graph, neighbor, visited)
```

非递归版

```
def dfs_iterative(graph, start):
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()
        if node not in visited:
            print(node, end=' ')
            visited.add(node)
            # 后入先出，所以逆序加入邻居
            stack.extend(reversed(graph[node]))
```

dijkstra算法

目的：用于求解最短路径问题

```
def dijkstra(graph, start, end):
    # graph: {u: [(v, weight), ...]}
    # start: 起点节点
    # 返回从 start 到所有其他节点的最短路径长度

    # 初始化距离表，所有点距离设为无穷大
    distances = {node: [float('inf'), None] for node in graph}
    distances[start] = [0, None]

    # 使用小根堆作为优先队列，元素是 (距离，当前点)
    queue = [(0, start)]

    while queue:
        current_dist, current_node = heapq.heappop(queue)
        if current_node == end:
            return current_dist
        # 如果当前节点的距离比记录的还大，说明是旧信息，跳过
        if current_dist > distances[current_node][0]:
            continue

        # 遍历当前节点的所有邻居
        for neighbor, weight in graph[current_node]:
            distance = current_dist + weight

            # 如果找到更短路径则更新，并加入优先队列
            if distance < distances[neighbor][0]:
                distances[neighbor] = [distance, current_node]
                heapq.heappush(queue, (distance, neighbor))

    return
```

Prim 算法（最小生成树）

```python
def prim_adj_list(n, adj, start=0):
    """
    使用邻接表和堆优化的 Prim 算法
    :param n: 节点总数
    :param adj: 邻接表，adj[u] 是一个列表，包含 (v, 权重) 元组
    :param start: 起始节点
    :return: 最小生成树的总权重和边列表 [(u, v, weight), ...]
    """
    visited = [False] * n
    min_heap = [(0, start, -1)]  # (权重, 当前节点, 来源节点)
    total_weight = 0
    mst_edges = []

    while min_heap:
        weight, u, parent = heapq.heappop(min_heap)
        if visited[u]:
            continue
        visited[u] = True
        total_weight += weight
        if parent != -1:
            mst_edges.append((parent, u, weight))
        for v, w in adj[u]:
            if not visited[v]:
                heapq.heappush(min_heap, (w, v, u))

    return total_weight, mst_edges
```

拓扑排序

目的：用于检查图是否有环，无环图可以给出排序并确定排序是否唯一

```python
def topological_sort(graph, num_vertices):
    # graph: {u: [(weight, v), ...]}
    in_degree = [0] * num_vertices
    for u in graph:
        for v, _ in graph[u]:
            in_degree[v] += 1

    queue = deque([i for i in range(num_vertices) if in_degree[i] == 0])
    topo_order = []

    while queue:
        u = queue.popleft()
        topo_order.append(u)
        for v, _ in graph[u]:
            in_degree[v] -= 1
            if in_degree[v] == 0:
                queue.append(v)
```

```
    if len(topo_order) != num_vertices:
        raise ValueError("图中存在环，无法进行拓扑排序")
    return topo_order
```

注意这里的实现要求node按照0 - num_vertices - 1编号