|  |  |  |
| --- | --- | --- |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное автономное  образовательное учреждение высшего образования  «Пермский государственный национальный  исследовательский университет» | | |
|  | Институт компьютерных наук и технологий | |
| **ОТЧЁТ**  по индивидуальной работе №2  по дисциплине «Язык программирования Python»  Вариант 17 | | |
|  | | Работу выполнил  студент группы ИТ-5,2024-2025 1 курса  Шалашов А. Ю.  «\_\_\_» \_\_\_\_\_\_\_\_2025 г. |
| Работу проверил  \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Фамилия И.О.  «\_\_\_» \_\_\_\_\_\_\_\_\_ 2025 г. |
| Пермь 2025 | | |

СОДЕРЖАНИЕ

[Постановка задачи 3](#__RefHeading___1)

[Алгоритм решения 5](#__RefHeading___2)

[Тестирование 7](#__RefHeading___3)

[Код программы 9](#__RefHeading___5)

# Постановка задачи

«Почтальон» Мудрый тритон работает почтальоном в Лукоморье. В перечень его обязанностей входит доставка корреспонденции птицам, разместившим свои гнезда на ветвях дуба (постепенно вытеснив оттуда русалку). Птицы вьют свои гнезда там, где ветка дерева разделяется на несколько более тонких веток. Лукоморье является районом с развитой инфраструктурой, поэтому все разветвления на дубе заняты гнездами птиц. За время работы тритон не только запомнил, где кто живет, но и уяснил, что если его путь проходит через чье-либо гнездо, и его обитатели не получают при этом корреспонденции, ему приходится извиняться за доставленные неудобства. Естественно, интеллигентный тритон стремится как можно меньше беспокоить обитателей дуба, поэтому он заранее планирует свой маршрут так, чтобы приносить минимальное число извинений.

Напишите программу, которая по описанию мест обитания птиц и списку корреспонденции определяет, сколько раз придется извиняться мудрому тритону.

Каждое гнездо дуба обозначено уникальным номером, что соответствует номеру квартиры, который мы иногда указываем, отправляя письма родным и близким по обычной почте. Гнездо, расположенное на первой развилке ствола имеет номер один. Тритон может только ползти по веткам или стволу дуба, прыжки и перелеты полностью исключены. Никакие ветки не соприкасаются и не пересекаются.

Формат входного файла: в первой строке входного файла находится число M – количество гнезд на дубе (0<M<1000). В следующих M строках находится информация о размещении гнезд и количестве писем для каждого из возможных адресатов. Первые два числа ni и li – количество соседей гнезда с номером i и количество направленных по этому адресу писем соответственно. Далее в строке приводится ni чисел – номера гнезд, являющихся соседними с гнездом c номером i.

Формат выходного файла: в единственной строке выходного файла должно содержаться единственное число – минимальное количество извинений, которые придется приносить мудрому тритону, для того чтобы доставить все письма.

Пример:

(входной файл)

5

3 2 2 5 4

1 1 1

1 1 4

2 2 3 1

1 3 1

(выходной файл)

2

# Алгоритм решения

**Построение дерева гнезд:**

Создаем структуру данных для представления дерева (список смежности)

Заполняем связи между гнездами на основе входных данных

Определяем родительские связи (для построения путей)

**Поиск путей доставки:**

Для каждого гнезда с письмами (li > 0) находим путь от корня (гнездо 1) до этого гнезда

Используем поиск в глубину (DFS) или информацию о родителях для восстановления пути

**Подсчет извинений:**

Проходим по всем гнездам на пути (кроме конечного целевого)

Если в гнезде нет писем (li == 0) - увеличиваем счетчик извинений

Если письма есть - пропускаем (извинения не нужны)

**Оптимизация:**

Запоминаем уже учтенные гнезда, чтобы не считать повторные проходы

Минимизируем количество операций, избегая повторных вычислений

**Выбранная структура данных**

Список смежности (реализованный как словарь): Идеально представляет древовидную структуру (ветвящиеся гнезда)

**Преимущества**

Быстрый доступ к соседним узлам (O(1)) Экономия памяти (хранит только существующие связи) Удобна для обходов (DFS/BFS)

**Словарь родителей**

Позволяет быстро восстанавливать пути от любого гнезда к корню

**Словарь писем**

Обеспечивает мгновенную проверку наличия писем в гнезде

Обработка исключений

**Чтение файла:** (файл не существует/нет прав)

**Валидация данных:** Гарантирует корректность структуры входных данных

**Запись результата:** Защита от проблем с записью (диск полон/нет прав)

**Классы:**

**Dubo:** Инкапсулирует логику работы с деревом

**UserInterface:** Отвечает за взаимодействие с пользователем

**Инкапсуляция:** Все поля классов (tree, letters и др.) приватные (по соглашению, через self.\_) Доступ только через методы класса

**Наследование и полиморфизм** не используются: В данной задаче нет:

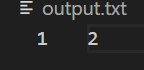
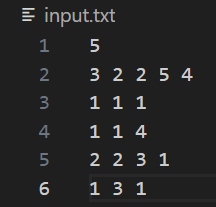
Иерархии объектов (все гнезда одинаковы)

Необходимости переопределять методы

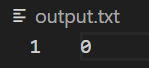
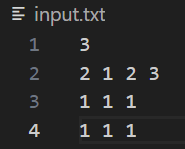
**Процедурный стиль в методах:** Алгоритм линейный и не требует сложной объектной модели

# Тестирование

1. Базовые значения (из условия задачи)

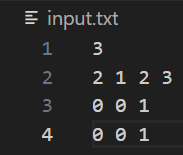


1. Все гнезда получают письма

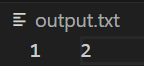
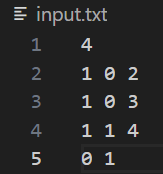


Все гнезда содержат письма, поэтому извинения не требуются.

1. Несоответствие в количестве соседей

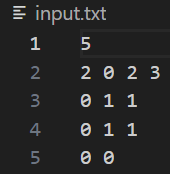


1. Письма на концах



Доставка в 4: путь 1→2→3→4. Извинения за проход через 2 и 3 (в 3 есть письма, но они не текущая цель).

1. Неправильное количество гнезд



# Код программы

**Ссылка на github:** https://github.com/Andrusbek1337/IKM\_Shalashov\_Andrew\_IT-5

class Dubo: # Класс, представляющий дерево гнезд и логику доставки писем.

def \_\_init\_\_(self): # Инициализация

self.tree = {}

self.letters = {}

self.parent = {}

self.paths = {}

def build\_tree(self, M, data): # Строит дерево гнезд

self.tree = {i: [] for i in range(1, M + 1)}

self.letters = {i: 0 for i in range(1, M + 1)}

self.parent = {1: None} # Гнездо 1 - корень дерева

for i in range(1, M + 1):

parts = data[i - 1]

ni = parts[0]

li = parts[1]

neighbors = parts[2:]

self.letters[i] = li

for neighbor in neighbors: # Строим связи между гнездами

if neighbor not in self.parent:

self.parent[neighbor] = i

self.tree[i].append(neighbor)

def compute\_paths(self): # Вычисляет пути от корня до каждого гнезда

for target in range(1, len(self.tree) + 1):

path = []

current = target

while current is not None:

path.append(current)

current = self.parent.get(current, None)

path.reverse()

self.paths[target] = path

def count\_apologies(self): # Подсчитывает количество необходимых извинений.

targets = {node for node in range(1, len(self.tree)+1)

if self.letters[node] > 0}

apologies\_needed = set()

for target in targets:

path = self.paths[target]

for node in path:

if node != target:

if self.letters[node] == 0 or node in targets:

apologies\_needed.add(node)

return len(apologies\_needed)

class UserInterface: # Класс для взаимодействия с пользователем и работы с файлами

def \_\_init\_\_(self): # Инициализация интерфейса пользователя

self.dubo = Dubo()

self.input\_filename = "input.txt"

self.output\_filename = "output.txt"

def read\_data(self): # Читает данные из входного файла

try:

with open(self.input\_filename, 'r') as file:

lines = [line.strip() for line in file if line.strip()]

M = int(lines[0]) # Первая строка - количество гнезд

data = []

for line in lines[1:M + 1]: # Остальные строки - данные о гнездах

parts = list(map(int, line.split()))

data.append(parts)

return M, data

except FileNotFoundError:

raise FileNotFoundError(f"Файл '{self.input\_filename}' не найден.")

except Exception as e:

raise ValueError(f"Ошибка при чтении файла: {e}")

def validate\_data(self, M, data): # Проверяет корректность входных данных

if M <= 0 or M >= 1000:

raise ValueError("Количество гнезд должно быть 0 < M < 1000.")

if len(data) != M:

raise ValueError("Количество строк данных не соответствует M.")

for i in range(M):

parts = data[i]

ni = parts[0] # Количество соседей

li = parts[1] # Количество писем

neighbors = parts[2:] # Список соседей

if ni != len(neighbors):

raise ValueError(f"Несоответствие количества соседей для гнезда {i+1}.")

if any(n < 1 or n > M for n in neighbors):

raise ValueError(f"Некорректный номер соседа для гнезда {i+1}.")

def process\_data(self, M, data): # Обрабатывает данные и возвращает результат

self.dubo.build\_tree(M, data)

self.dubo.compute\_paths()

return self.dubo.count\_apologies()

def write\_output(self, result): # Записывает результат в файл

try:

with open(self.output\_filename, 'w') as file:

file.write(str(result))

print(f"Результат записан в '{self.output\_filename}'.")

except Exception as e:

raise IOError(f"Ошибка записи: {e}")

def run(self): # Основной метод запуска программы

print("Программа для подсчета извинений мудрого тритона.")

try:

M, data = self.read\_data() # 1. Чтение данных

self.validate\_data(M, data) # 2. Проверка данных

result = self.process\_data(M, data) # 3. Обработка и получение результата

self.write\_output(result) # 4. Запись результата

except FileNotFoundError as e:

print(f"Ошибка: {e}")

except ValueError as e:

print(f"Ошибка данных: {e}")

except Exception as e:

print(f"Неожиданная ошибка: {e}")

if \_\_name\_\_ == '\_\_main\_\_': # Создаем и запускаем интерфейс пользователя

ui = UserInterface()

ui.run()