САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №3

по курсу «Алгоритмы и структуры данных»

Вариант 1

Выполнил:

Азаренков Георгий Денисович

К34421

Проверил:

Афанасьев А.В.

Санкт-Петербург

2024 г.

# Содержание отчета

[Содержание отчета 2](#_Toc179019075)

[Комментарий к общему коду 3](#_Toc179019076)

[Задача 1 5](#_Toc179019077)

[Текст задачи 5](#_Toc179019078)

[Код решения 7](#_Toc179019079)

[Результаты выполнения 7](#_Toc179019080)

[Задача 5 7](#_Toc179019081)

[Текст задачи 7](#_Toc179019082)

[Код решения 7](#_Toc179019083)

[Результаты выполнения 7](#_Toc179019084)

[Задача 6 8](#_Toc179019085)

[Текст задачи 8](#_Toc179019086)

[Код решения 8](#_Toc179019087)

[Результаты выполнения 8](#_Toc179019088)

[Вывод 10](#_Toc179019089)

# Комментарий к общему коду

При использовании всех задач я использую код из файла “common.py”, который находится в корне репозитория.

from typing import List, Callable  
  
  
def solve(solver: Callable[[List[str]], List[str]]) -> None:  
 with open("../../input.txt", "r") as file:  
 lines = file.readlines()  
 input\_lines = list(map(lambda x: x.strip(), lines))  
  
 output\_lines = solver(input\_lines)  
  
 with open("../../output.txt", "w") as file:  
 for line in output\_lines:  
 file.write(f"{line}\n")

Использование функции “solve” позволило мне избавиться от необходимости раз за разом имплементировать логику чтения и сохранения данных.

# Задача 1

## Текст задачи

A paper with text and numbers

Description automatically generated

## Код решения

from collections import deque  
from typing import List  
  
from common import solve  
  
"""   
Функция проверяет, существует ли путь между двумя заданными вершинами в неориентированном графе.  
Принимает список строк, представляющих входные данные, и возвращает список строк с результатом.  
"""  
  
  
def get\_solution(input\_lines: List[str]) -> List[str]:  
 *# Разбор первой строки для получения количества вершин и рёбер* n, m = map(int, input\_lines[0].split())  
  
 *# Инициализация списка смежности* adjacency\_list = {i: [] for i in range(1, n + 1)}  
  
 *# Построение графа из списка рёбер* for i in range(1, m + 1):  
 a, b = map(int, input\_lines[i].split())  
 adjacency\_list[a].append(b)  
 adjacency\_list[b].append(a)  
  
 *# Получение вершин u и v для проверки пути* u, v = map(int, input\_lines[m + 1].split())  
  
 queue = deque()  
 visited = set()  
  
 *# Начало поиска с вершины u* queue.append(u)  
 visited.add(u)  
  
 *# Выполнение BFS для поиска пути до v* while queue:  
 current = queue.popleft()  
  
 if current == v:  
 return ["1"]  
  
 for neighbor in adjacency\_list[current]:  
 if neighbor not in visited:  
 visited.add(neighbor)  
 queue.append(neighbor)  
  
 *# Если путь не найден* return ["0"]  
  
  
solve(get\_solution)

## Результаты выполнения

input.txt

A screenshot of a computer

Description automatically generated

output.txt

A close up of a calendar

Description automatically generated

Пример 2

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

# Задача 5

## Текст задачи

A paper with text and numbers

Description automatically generated

## Код решения

from typing import List, Dict  
  
from common import solve  
  
"""   
Функция get\_solution принимает список строк, представляющих входные данные,   
и возвращает список строк с результатом – количеством компонент сильной связности.  
"""  
  
  
*# Функция для выполнения DFS и заполнения порядка завершения*def dfs\_first(u: int, visited: List[bool], stack: List[int], adj: Dict[int, List[int]]) -> None:  
 visited[u] = True  
  
 for v in adj[u]:  
 if not visited[v]:  
 dfs\_first(v, visited, stack, adj)  
  
 stack.append(u)  
  
  
*# Функция для выполнения DFS на транспонированном графе*def dfs\_second(u: int, visited: List[bool], adj\_transpose: Dict[int, List[int]]) -> None:  
 visited[u] = True  
  
 for v in adj\_transpose[u]:  
 if not visited[v]:  
 dfs\_second(v, visited, adj\_transpose)  
  
  
def get\_solution(input\_lines: List[str]) -> List[str]:  
 *# Разбор первой строки для получения количества вершин и ребер* n, m = map(int, input\_lines[0].split())  
  
 *# Создание списка смежности для графа* adj: Dict[int, List[int]] = {i: [] for i in range(1, n + 1)}  
  
 *# Заполнение списка смежности ребрами из входных данных* for line in input\_lines[1 : m + 1]:  
 u, v = map(int, line.split())  
 adj[u].append(v)  
  
 *# Транспонирование графа* adj\_transpose: Dict[int, List[int]] = {i: [] for i in range(1, n + 1)}  
  
 for u in adj:  
 for v in adj[u]:  
 adj\_transpose[v].append(u)  
  
 *# Первый проход DFS для заполнения стека* stack: List[int] = []  
 visited: List[bool] = [False] \* (n + 1)  
  
 for u in range(1, n + 1):  
 if not visited[u]:  
 dfs\_first(u, visited, stack, adj)  
  
 *# Второй проход DFS на транспонированном графе* visited = [False] \* (n + 1)  
 count = 0  
  
 while stack:  
 u = stack.pop()  
  
 if not visited[u]:  
 dfs\_second(u, visited, adj\_transpose)  
 count += 1  
  
 *# Возврат результата в виде списка строк* return [str(count)]  
  
  
solve(get\_solution)

## Результаты выполнения

input.txt

A screenshot of a computer

Description automatically generated

output.txt

A calendar with numbers and letters

Description automatically generated

Пример 2

A screenshot of a computer

Description automatically generated

A close up of a number

Description automatically generated

# Задача 6

## Текст задачи

A paper with text and numbers

Description automatically generated

## Код решения

from collections import deque  
from typing import List  
  
from common import solve  
  
  
def get\_solution(input\_lines: List[str]) -> List[str]:  
 *"""  
 Кратчайший путь между двумя городами в неориентированном графе.  
 """  
  
 # Разбираем первую строку, получаем количество вершин и ребер* n, m = map(int, input\_lines[0].split())  
  
 *# Инициализируем список смежности для представления графа* graph = [[] for \_ in range(n + 1)] *# Используем n + 1 для удобства индексации вершин с 1  
  
 # Заполняем граф на основе последующих m строк* for i in range(1, m + 1):  
 a, b = map(int, input\_lines[i].split())  
 graph[a].append(b)  
 graph[b].append(a)  
  
 *# Извлекаем вершины u и v, между которыми нужно найти кратчайший путь* u, v = map(int, input\_lines[m + 1].split())  
  
 *# Массив для хранения расстояний от вершины u до всех остальных* distance = [-1] \* (n + 1)  
 distance[u] = 0 *# Расстояние до себя равно нулю  
  
 # Очередь для обхода графа в ширину (BFS)* queue = deque()  
 queue.append(u)  
  
 *# BFS для поиска кратчайшего пути* while queue:  
 current = queue.popleft()  
  
 for neighbor in graph[current]:  
 if distance[neighbor] == -1:  
 distance[neighbor] = distance[current] + 1  
 queue.append(neighbor)  
  
 *# Получаем длину кратчайшего пути до вершины v* result = distance[v]  
  
 *# Если пути нет, distance[v] останется равным -1  
  
 # Возвращаем результат в виде списка строк* return [str(result)]  
  
  
solve(get\_solution)

## Результаты выполнения

Пример 1

input.txt

A screenshot of a computer

Description automatically generated

output.txt

A screenshot of a calendar

Description automatically generated

Пример 2

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

# Вывод

В ходе данной лабораторной работы я научился решать задачи по теме. Написанные программы были протестированы с измерением затрат времени и оперативной памяти. Все программы работаю корректно и укладываются в установленные ограничения.