

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»**

Кафедра теоретических основ
компьютерной безопасности и
криптографии

Практическая работа

**ОТЧЁТ ПО ДИСЦИПЛИНЕ
«НЕЙРОННЫЕ СЕТИ»**

студентки 5 курса 531 группы

специальности 10.05.01 Компьютерная безопасность

факультета компьютерных наук и информационных технологий

Зиминой Ирины Олеговны

Преподаватель

доцент

подпись, дата

И.И. Слеповичев

Саратов 2023

Задание 1. Создание ориентированного графа

На входе: текстовый файл с описанием графа в виде списка дуг:

$$(a_1, b_1, n_1), (a_2, b_2, n_2), \dots, (a_k, b_k, n_k),$$

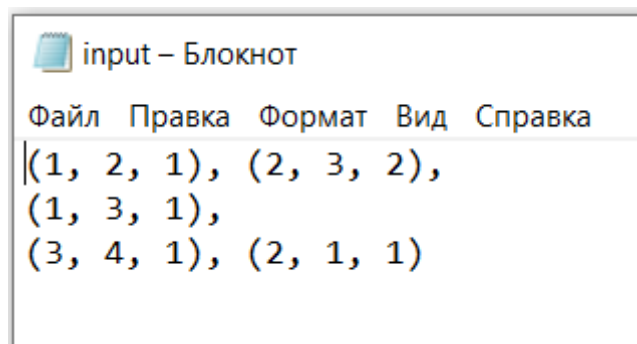
где a_i – начальная вершина дуги i , b_i – конечная вершина дуги i , n_i – порядковый номер дуги в списке всех заходящих в вершину b_i дуг.

На выходе: ориентированный граф с именованными вершинами и линейно упорядоченными дугами (в соответствии с порядком из текстового файла).

Если присутствует ошибка, то в выводе: сообщение об ошибке в формате файла.

Пример работы программы.

Входной список дуг:



```
input - Блокнот
Файл  Правка  Формат  Вид  Справка
(1, 2, 1), (2, 3, 2),
(1, 3, 1),
(3, 4, 1), (2, 1, 1)
```

Запуск программы командой:

```
python task1.py -i "input.txt" -o "output.xml"
```

Результат работы программы:

```

▼<graph>
  <vertex>v1</vertex>
  <vertex>v2</vertex>
  <vertex>v3</vertex>
  <vertex>v4</vertex>
  ▼<arc>
    <from>v1</from>
    <to>v2</to>
    <order>1</order>
  </arc>
  ▼<arc>
    <from>v1</from>
    <to>v3</to>
    <order>1</order>
  </arc>
  ▼<arc>
    <from>v2</from>
    <to>v3</to>
    <order>2</order>
  </arc>
  ▼<arc>
    <from>v2</from>
    <to>v1</to>
    <order>1</order>
  </arc>
  ▼<arc>
    <from>v3</from>
    <to>v4</to>
    <order>1</order>
  </arc>
</graph>

```

Задание 2. Создание функции по графу

На входе: ориентированный граф с именованными вершинами, как описано в задании 1.

На выходе: линейное представление, реализуемое графом в префиксной скобочной записи: $A_1(B_1(C_1(\dots), \dots, C_m(\dots)), \dots, B_n(\dots))$.

Пример работы программы.

Входной список дуг:

input2 – Блокнот

Файл Правка Формат Вид Справка

(2, 3, 1), (1, 3, 2), (3, 4, 1), (1, 5, 1), (6, 5, 3), (3, 5, 2), (2, 3, 3)

Запуск программы командой:

python task2.py -i "input2.txt" -o "output2.txt"

Результат работы программы:

output2 – Блокнот

Файл Правка Формат Вид Справка

4(3(2, 1, 2)), 5(1, 3(2, 1, 2), 6)

Задание 3. Вычисление значения функции на графе

На входе:

- Текстовый файл с описанием в виде списка дуг (согласно 1-му заданию).
- Тестовый файл соответствий арифметических операций вершин:

```
{  
    a1: операция_1  
    a2: операция_2  
    ...  
    an: операция_n  
}
```

где a_i – имя i -й вершины, операция_ i – символ операции,
соответствующий вершине a_i .

Допустимы следующие символы операций:

+ – сумма значений,

* – произведение значений,

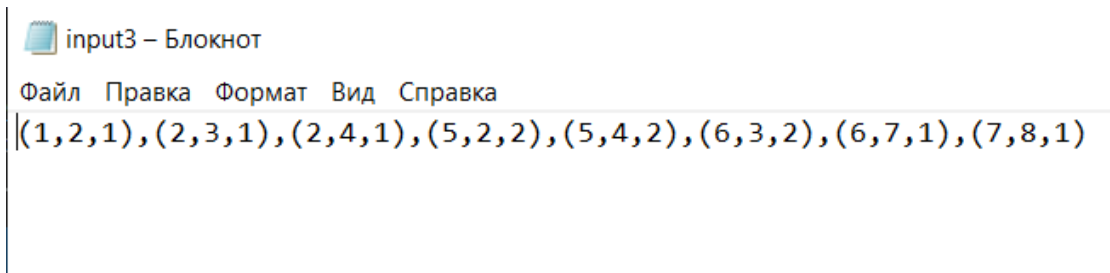
exp – экспонирование входного значения,

число – любая числовая константа.

На выходе: значение функции, построенной по графу а) и файлу b).

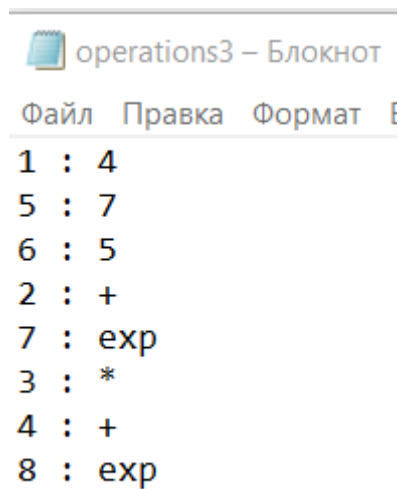
Пример работы программы

Входной список дуг:



```
input3 – Блокнот
Файл  Правка  Формат  Вид  Справка
(1,2,1), (2,3,1), (2,4,1), (5,2,2), (5,4,2), (6,3,2), (6,7,1), (7,8,1)
```

Входной файл с операциями:

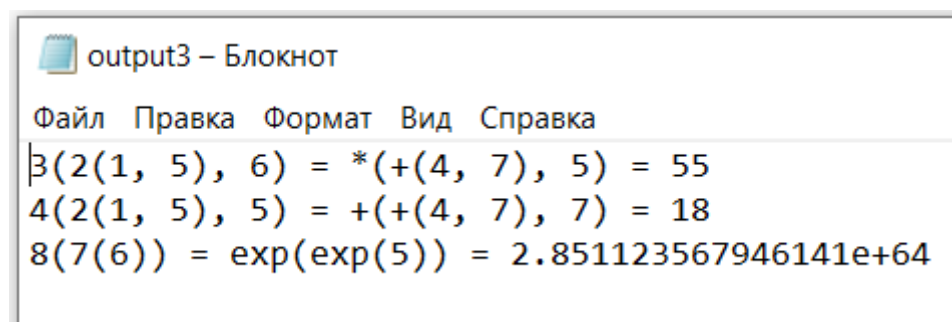


```
operations3 – Блокнот
Файл  Правка  Формат  {
1 : 4
5 : 7
6 : 5
2 : +
7 : exp
3 : *
4 : +
8 : exp
```

Запуск программы командой:

```
python task3.py -i "input3.txt" --operations operations3.tx
-o "output3.txt"
```

Результат работы программы:



```
output3 – Блокнот
Файл  Правка  Формат  Вид  Справка
3(2(1, 5), 6) = *(+(4, 7), 5) = 55
4(2(1, 5), 5) = +(+(4, 7), 7) = 18
8(7(6)) = exp(exp(5)) = 2.851123567946141e+64
```

Приложение А

Код задания 1. task1.py

```
import logging
import argparse
import xml.etree.cElementTree as ET
from xml.dom import minidom

logger = logging.getLogger(__name__)

class InputException(Exception):
    """Exception raised for errors in the input file.

    Attributes:
        input_file -- input file's name
        line -- line that exception raised
    """

    def __init__(self, input_file, line):
        self.input_file = input_file
        self.line = line
        super().__init__()

class DataException(Exception):
    """Exception raised for logic errors .

    Attributes:
        input_file -- input file's name
        line -- line that exception raised
        message -- explanation of the error
    """

    def __init__(self, input_file, line, message):
        self.input_file = input_file
        self.line = line
        self.message = message
        super().__init__(message)

def validate_input_data(lines, input_file_name):
    for j in range(len(lines)):
        if '-' in lines[j]:
            raise InputException(input_file_name, j + 1)

        for e in lines[j]:
            if e.isalpha():
                raise InputException(input_file_name, j + 1)

    return True

def read_graph(input_file_name):
```

```

with open(input_file_name, 'r') as input_graph:
    edges = {}

    lines = input_graph.read()
    lines = lines.replace(' ', '')
    lines = lines.split('\n')

    validate_input_data(lines, input_file_name)

    es = ''.join(lines)
    es = es.split('), (')
    max_vertex = max(int(es[0][1]), int(es[0][3]))

    for i in range(1, len(es)):
        max_vertex = max(max_vertex, int(es[i][0]), int(es[i][2]))

    in_number = [""] * max_vertex

    for i in range(len(es)):
        if i == 0:
            es[i] = es[i][1:]
        if i == len(es) - 1:
            if es[i][len(es[i]) - 1] == '\n':
                es[i] = es[i][:len(es[i]) - 2]
            else:
                es[i] = es[i][:-1]
        try:
            edge = eval(es[i])
            if len(edge) != 3:
                edge = str(edge).replace(" ", '')
                for j in range(len(lines)):
                    if edge in lines[j]:
                        raise InputException(input_file_name, j + 1)
        except SyntaxError:
            edge = str(edge).replace(" ", '')
            for j in range(len(lines)):
                if edge in lines[j]:
                    raise InputException(input_file_name, j + 1)

        if edge[0] not in edges:
            edges[edge[0]] = []
        if edge[1] not in edges:
            edges[edge[1]] = []

        in_number[edge[1] - 1] = in_number[edge[1] - 1] + " " +
str(edge[2])
        edges[edge[0]].append([edge[2], edge[1]])

    y = []

    for x in in_number:
        y.append(x.split(" "))
    for i in range(len(y)):
        num = []
        for j in range(1, len(y[i])):
            num.append(int(y[i][j]))
        num.sort()
        if len(num) == 1 and num[0] != 1:

```

```

        raise DataException(input_file_name, len(lines),
'Неправильная нумерация')
    for j in range(0, len(num) - 1):
        if num[j] == num[j + 1]:
            raise DataException(input_file_name, len(lines),
'Неправильная нумерация')
        if num[j + 1] - num[j] != 1:
            raise DataException(input_file_name, len(lines),
'Неправильная нумерация')
    return edges

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('-i', '--input', required=True, help='Имя
входного файла')
    parser.add_argument('-o', '--output', help='Имя выходного файла')
    parser.add_argument('--log-file', help='Имя файла с логом
программы', dest='log_file')
    parser.add_argument('--log-level', help='Уровень логирования',
dest='log_level', default='debug')

    args = parser.parse_args()

    numeric_level = getattr(logging, args.log_level.upper(), None)
    if not isinstance(numeric_level, int):
        raise ValueError('Invalid log level: %s' % args.log_level)

    logging.basicConfig(level=numeric_level, filename=args.log_file,
encoding='utf-8')

    try:
        all_edges = read_graph(args.input)

        root = ET.Element("graph")

        for x in range(len(all_edges)):
            ET.SubElement(root, "vertex").text = 'v' + str(x + 1)
        for x in range(1, len(all_edges)):
            for z in range(len(all_edges[x])):
                arc = ET.SubElement(root, "arc")
                From = ET.SubElement(arc, 'from').text = 'v' + str(x)
                To = ET.SubElement(arc, 'to').text = 'v' +
str(all_edges[x][z][1])
                Order = ET.SubElement(arc, 'order').text =
str(all_edges[x][z][0])

            dom = minidom.parseString(ET.tostring(root))
            tree = dom.toprettyxml(indent='\t')

            if args.output is not None:
                with open(args.output, 'w') as file:
                    file.write(tree)
            else:
                print(tree)
        except InputException as e:
            logging.fatal("Ошибка в данных входного файла %s в строке %s",
e.input_file, e.line)
        except DataException as e:

```



```
        logging.fatal("Ошибка в логике данных входного файла %s в строке  
%s. Текст ошибки %s", e.input_file, e.line, e.message)  
    except Exception as e:  
        logging.fatal("Неизвестная ошибка")  
        logging.exception(e)  
  
if __name__ == '__main__':  
    main()
```

Приложение Б

Код задания 2. Task2.py

```
import argparse
import logging

logger = logging.getLogger(__name__)

class InputException(Exception):
    """Exception raised for errors in the input file.

    Attributes:
        input_file -- input file's name
        line -- line that exception raised
    """

    def __init__(self, input_file, line):
        self.input_file = input_file
        self.line = line

        super().__init__()

class DataException(Exception):
    """Exception raised for logic errors .

    Attributes:
        input_file -- input file's name
        line -- line that exception raised
        message -- explanation of the error
    """

    def __init__(self, input_file, line, message):
        self.input_file = input_file
        self.line = line
        self.message = message

        super().__init__(message)

class CycleException(Exception):
    """Exception raised when cycle found .

    Attributes:
        v -- first vertex
        u -- second vertex
        message -- explanation of the error
    """

    def __init__(self, v, u):
        self.v = v
        self.u = u

        super().__init__()
```

```

def validate_input_data(lines, input_file_name):
    for j in range(len(lines)):
        if '-' in lines[j]:
            raise InputException(input_file_name, j + 1)

        for e in lines[j]:
            if e.isalpha():
                raise InputException(input_file_name, j + 1)

    return True

def graph_reading_in_format(input_file_name):
    with open(input_file_name, 'r') as input_graph:
        edges = {}
        o_i_list = {}
        lines = input_graph.read()
        lines = lines.replace(' ', '')
        lines = lines.split('\n')

        validate_input_data(lines, input_file_name)

        es = ''.join(lines)
        es = es.split('), (')

        max_vertex = max(int(es[0][1]), int(es[0][3]))

        for i in range(1, len(es)):
            max_vertex = max(max_vertex, int(es[i][0]), int(es[i][2]))

        in_number = [""] * max_vertex

        for i in range(len(es)):
            if i == 0:
                es[i] = es[i][1:]
            if i == len(es) - 1:
                if es[i][len(es[i]) - 1] == '\n':
                    es[i] = es[i][:len(es[i]) - 2]
                else:
                    es[i] = es[i][:-1]

            try:
                edge = eval(es[i])
                if len(edge) != 3:
                    edge = str(edge).replace(" ", '')
                    for j in range(len(lines)):
                        if edge in lines[j]:
                            raise InputException(input_file_name, j + 1)
            except SyntaxError:
                edge = str(edge).replace(" ", '')
                for j in range(len(lines)):
                    if edge in lines[j]:
                        raise InputException(input_file_name, j + 1)

            if edge[0] not in edges:
                edges[edge[0]] = []
                o_i_list[edge[0]] = [1, 0]

```

```

        else:
            o_i_list[edge[0]][0] += 1

        if edge[1] not in edges:
            edges[edge[1]] = []
            o_i_list[edge[1]] = [0, 1]
        else:
            o_i_list[edge[1]][1] += 1

        in_number[edge[1] - 1] = in_number[edge[1] - 1] + " " +
str(edge[2])
        edges[edge[0]].append([edge[2], edge[1]])

y = []
for x in in_number:
    y.append(x.split(" "))
for i in range(len(y)):
    num = []
    for j in range(1, len(y[i])):
        num.append(int(y[i][j]))

    num.sort()

    if len(num) == 1 and num[0] != 1:
        raise DataException(input_file_name, len(lines), 'Неправильная
нумерация')

    for j in range(0, len(num) - 1):
        if num[j] == num[j + 1]:
            raise DataException(input_file_name, len(lines),
'Неправильная нумерация')
        if num[j + 1] - num[j] != 1:
            raise DataException(input_file_name, len(lines),
'Неправильная нумерация')
    graph = {}

    for j in range(1, len(edges) + 1):
        graph[j] = []
        for i in range(len(edges[j])):
            graph[j].append([edges[j][i][0], edges[j][i][1]])

    return graph, o_i_list

def construct_by_dfs(v, graph):
    some = False
    first = True
    global output

    for vertex in graph[v]:
        if not first:
            output += ", "
        some = True
        if first:
            output += str(v)
            output += "("
            first = False
        construct_by_dfs(vertex[1], graph)

```

```

    if some:
        output += ")"
    if not some:
        if first:
            output += str(v)

    return output

def coloring(v, graph, visited, parts):
    visited[v] = True
    dfs_graph = {v: []}

    for vertex in graph[v]:
        if vertex[1] not in visited:
            dfs_graph[v].append(coloring(vertex[1], graph, visited, parts))
        else:
            if vertex[1] not in parts:
                raise CycleException(v, vertex[1])
            else:
                dfs_graph[v].append({vertex[1]: parts[vertex[1]]})

    parts[v] = dfs_graph[v]
    return dfs_graph

def cycle_validator(graph, d):
    started_vertex = []
    for vertex in d:
        if d[vertex][1] == 0:
            started_vertex.append(vertex)

    parts = {}
    visited = {}
    for vertex in started_vertex:
        coloring(vertex, graph, visited, parts)

def cycle_finding(graph, d):
    started_vertex = []
    for vertex in d:
        if d[vertex][1] == 0:
            started_vertex.append(vertex)

    if not started_vertex:
        raise CycleException(1, len(d))

    global output
    output = ""

    for vertex in range(len(started_vertex)):
        output = construct_by_dfs(started_vertex[vertex], graph)
        if vertex != len(started_vertex) - 1:
            output += ", "

    return output

```

```

def get_reverse_graph(graph):
    reversed_graph = {}
    for u in sorted(graph.keys()):
        for v in graph[u]:
            if u not in reversed_graph:
                reversed_graph[u] = []
            if v[1] not in reversed_graph:
                reversed_graph[v[1]] = []
            reversed_graph[v[1]].append([v[0], u])

    return reversed_graph

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('-i', '--input', required=True, help='Имя входного файла')
    parser.add_argument('-o', '--output', help='Имя выходного файла')
    parser.add_argument('--log-file', help='Имя файла с логом программы', dest='log_file')
    parser.add_argument('--log-level', help='Уровень логирования', dest='log_level', default='debug')

    args = parser.parse_args()

    numeric_level = getattr(logging, args.log_level.upper(), None)
    if not isinstance(numeric_level, int):
        raise ValueError('Invalid log level: %s' % args.log_level)

    logging.basicConfig(level=numeric_level, filename=args.log_file, encoding='utf-8')

    try:
        all_edges, o_i_list = graph_reading_in_format(args.input)

        cycle_validator(all_edges, o_i_list)

        for x in range(1, len(o_i_list) + 1):
            tmp = o_i_list[x][0]
            o_i_list[x][0] = o_i_list[x][1]
            o_i_list[x][1] = tmp
        reverse = get_reverse_graph(all_edges)
        graph = [[] * (len(all_edges) + 1)
        k = 0
        list = []
        for x in sorted(reverse.keys()):
            for i in range(len(reverse[x])):
                list.append([reverse[x][i]])
            graph[k + 1] = list
            k += 1
            list = []

        new_graph = []

        for i in graph:
            tmp = []

            for j in range(len(i)):

```

```

        tmp.append([i[j][0][0], i[j][0][1]])

    tmp.sort(key=lambda x: x[0])
    new_graph.append(tmp)

    output = cycle_finding(new_graph, o_i_list)

    if args.output is not None:
        with open(args.output, 'w') as file:
            file.write(output)
    else:
        print(output)
    except InputException as e:
        logging.fatal("Ошибка в данных входного файла %s в строке %s",
e.input_file, e.line)
    except DataException as e:
        logging.fatal("Ошибка в логике данных входного файла %s в строке %s.
Текст ошибки %s", e.input_file, e.line, e.message)
    except CycleException as e:
        logging.fatal("Существует цикл между вершинами %s и %s", e.v, e.u)
    except Exception as e:
        logging.fatal("Неизвестная ошибка")
        logging.exception(e)

if __name__ == '__main__':
    main()

```

Приложение С

Код задания 3. Task3.py

```
import argparse
import logging
import math

logger = logging.getLogger(__name__)

class InputException(Exception):
    '''Exception raised for errors in the input file.

    Attributes:
        input_file -- input file's name
        line -- line that exception raised
    '''

    def __init__(self, input_file, line):
        self.input_file = input_file
        self.line = line

        super().__init__()

class DataException(Exception):
    '''Exception raised for logic errors .

    Attributes:
        input_file -- input file's name
        line -- line that exception raised
        message -- explanation of the error
    '''

    def __init__(self, input_file, line, message):
        self.input_file = input_file
        self.line = line
        self.message = message

        super().__init__(message)

class CycleException(Exception):
    '''Exception raised when cycle found .

    Attributes:
        v -- first vertex
        u -- second vertex
        message -- explanation of the error
    '''

    def __init__(self, v, u):
        self.v = v
        self.u = u

        super().__init__()

class OperationFormatException(Exception):
    '''Exception raised due incorrect operation's format .

    Attributes:
```



```

        input_file -- input file's name
        line -- line that exception raised
        message -- explanation of the error
'''

def __init__(self, input_file, line, message):
    self.input_file = input_file
    self.line = line
    self.message = message

    super().__init__(message)

def validate_input_data(lines, input_file_name):
    for j in range(len(lines)):
        if '-' in lines[j]:
            raise InputException(input_file_name, j + 1)

        for e in lines[j]:
            if e.isalpha():
                raise InputException(input_file_name, j + 1)

    return True

def graph_reading_in_format(input_file_name):
    with open(input_file_name, 'r') as input_graph:
        edges = {}
        o_i_list = {}
        lines = input_graph.read()
        lines = lines.replace(' ', '')
        lines = lines.split('\n')

        validate_input_data(lines, input_file_name)

        es = ''.join(lines)
        es = es.split('), (')

        max_vertex = max(int(es[0][1]), int(es[0][3]))

        for i in range(1, len(es)):
            max_vertex = max(max_vertex, int(es[i][0]), int(es[i][2]))

        in_number = [''] * max_vertex

        for i in range(len(es)):
            if i == 0:
                es[i] = es[i][1:]
            if i == len(es) - 1:
                if es[i][len(es[i]) - 1] == '\n':
                    es[i] = es[i][:len(es[i]) - 2]
                else:
                    es[i] = es[i][:-1]

            try:
                edge = eval(es[i])
                if len(edge) != 3:
                    edge = str(edge).replace(' ', '')
                    for j in range(len(lines)):
                        if edge in lines[j]:
                            raise InputException(input_file_name, j + 1)
            except SyntaxError:
                edge = str(edge).replace(' ', '')

```

```

        for j in range(len(lines)):
            if edge in lines[j]:
                raise InputException(input_file_name, j + 1)

    if edge[0] not in edges:
        edges[edge[0]] = []
        o_i_list[edge[0]] = [1, 0]
    else:
        o_i_list[edge[0]][0] += 1

    if edge[1] not in edges:
        edges[edge[1]] = []
        o_i_list[edge[1]] = [0, 1]
    else:
        o_i_list[edge[1]][1] += 1

    in_number[edge[1] - 1] = in_number[edge[1] - 1] + ' ' + str(edge[2])
    edges[edge[0]].append([edge[2], edge[1]])

y = []
for x in in_number:
    y.append(x.split(' '))
for i in range(len(y)):
    num = []
    for j in range(1, len(y[i])):
        num.append(int(y[i][j]))

    num.sort()

    if len(num) == 1 and num[0] != 1:
        raise DataException(input_file_name, len(lines), 'Неправильная
нумерация')

    for j in range(0, len(num) - 1):
        if num[j] == num[j + 1]:
            raise DataException(input_file_name, len(lines), 'Неправильная
нумерация')
        if num[j + 1] - num[j] != 1:
            raise DataException(input_file_name, len(lines), 'Неправильная
нумерация')
    graph = {}

    for j in range(1, len(edges) + 1):
        graph[j] = []
        for i in range(len(edges[j])):
            graph[j].append([edges[j][i][0], edges[j][i][1]])

    return graph, o_i_list

def operations_reading_in_format(all_edges, input_operation_filename):
    operations = {}
    available_operations = ['+', '*', 'exp']
    j = 0
    with open(input_operation_filename, 'r') as input_operations:
        for line in input_operations:
            line = line[:len(line) - 1]
            line = line.replace(' ', '')
            pos = line.find(':')
            if pos == -1:
                raise OperationFormatException(input_operation_filename, j + 1,
'Ошибка ввода операции - не найден разделитель \':\'')

```

```

        vertex = int(line[:pos])
        if vertex not in all_edges:
            raise OperationFormatException(input_operation_filename, j + 1,
f'Ошибка ввода операции - в графе не существует такая вершина \'{vertex}\''')

        operation = str(line[(pos + 1):])
        try:
            if operation not in available_operations:
                operations[str(vertex)] = int(operation)
            else:
                operations[str(vertex)] = operation
        except:
            raise OperationFormatException(input_operation_filename, j + 1,
f'Ошибка ввода операции \'{operation}\'' - неверный формат числа или символа операции.
Проверьте что что строка не пустая и содержит корректные символы')
            j += 1

    return operations

def construct_by_dfs(v, graph):
    some = False
    first = True

    global output
    global tmp

    for vertex in graph[v]:
        if not first:
            output += ', '
            tmp += ', '
        some = True
        if first:
            output += str(v)
            tmp += str(v)
            output += '('
            tmp += '('
            first = False
        construct_by_dfs(vertex[1], graph)

    if some:
        output += ')'
        tmp += ')'

    if not some:
        if first:
            output += str(v)
            tmp += str(v)
    return output, tmp

def coloring(v, graph, visited, parts):
    visited[v] = True
    dfs_graph = {v: []}

    for vertex in graph[v]:
        if vertex[1] not in visited:
            dfs_graph[v].append(coloring(vertex[1], graph, visited, parts))
        else:
            if vertex[1] not in parts:
                raise CycleException(v, vertex[1])
            else:
                dfs_graph[v].append({vertex[1]: parts[vertex[1]]})

```

```

parts[v] = dfs_graph[v]
return dfs_graph

def cycle_validator(graph, d):
    started_vertex = []
    for vertex in d:
        if d[vertex][1] == 0:
            started_vertex.append(vertex)
    parts = {}
    visited = {}
    for vertex in started_vertex:
        coloring(vertex, graph, visited, parts)

def cycle_finding(graph, d):
    started_vertex = []
    for vertex in d:
        if d[vertex][1] == 0:
            started_vertex.append(vertex)
    if not started_vertex:
        raise CycleException(1, len(d))

    global output
    global fun
    global tmp

    tmp = ''
    fun = []
    output = ''

    for vertex in range(len(started_vertex)):
        output, tmp = construct_by_dfs(started_vertex[vertex], graph)
        if vertex != len(started_vertex) - 1:
            output += ', '
            fun.append(tmp)
            tmp = ''
        fun.append(tmp)
    return output, fun

def get_reverse_graph(graph):
    reversed_graph = {}
    for u in sorted(graph.keys()):
        for v in graph[u]:
            if u not in reversed_graph:
                reversed_graph[u] = []
            if v[1] not in reversed_graph:
                reversed_graph[v[1]] = []
            reversed_graph[v[1]].append([v[0], u])

    return reversed_graph

def dfs_operations(v, graph, operations, visited, values):
    visited[v] = True
    new_values = {v: -1}

    if type(operations[str(v)]) == int:
        new_values[v] = operations[str(v)]
    elif operations[str(v)] == '+':
        new_values[v] = 0

```

```

elif operations[str(v)] == '*' or operations[str(v)] == 'exp':
    new_values[v] = 1

if not len(graph[v]):
    values[v] = new_values[v]
    return values

for vertex in graph[v]:
    if vertex[1] not in visited:
        val = dfs_operations(vertex[1], graph, operations, visited, values)
        if operations[str(v)] == '+':
            new_values[v] += val[vertex[1]]
        elif operations[str(v)] == '*':
            new_values[v] *= val[vertex[1]]
        elif operations[str(v)] == 'exp':
            new_values[v] = math.exp(val[vertex[1]])
    else:
        if operations[str(v)] == '+':
            new_values[v] += values[vertex[1]]
        elif operations[str(v)] == '*':
            new_values[v] *= values[vertex[1]]
        elif operations[str(v)] == 'exp':
            new_values[v] = math.exp(values[vertex[1]])
values[v] = new_values[v]
return values

def do_eval_operation(graph, started_vertex, operations):
    visited = {}
    values = {}
    for v in started_vertex:
        dfs_operations(v, graph, operations, visited, values)
    return values

def check_operation_correctness(graph, operations, input_operation_file_name):
    for vertex in graph.keys():
        if type(operations[str(vertex)]) == int:
            if len(graph[vertex]) == 0:
                continue
            raise OperationFormatException(input_operation_file_name, '', f'Операция \'{operations[str(vertex)]}\'' не соответствует вершине \'{vertex}\''')
        elif operations[str(vertex)] == '+' or operations[str(vertex)] == '*':
            if len(graph[vertex]) > 1:
                continue
            raise OperationFormatException(input_operation_file_name, '', f'Операция \'{operations[str(vertex)]}\'' не соответствует вершине \'{vertex}\''')
        elif operations[str(vertex)] == 'exp':
            if len(graph[vertex]) == 1:
                continue
            raise OperationFormatException(input_operation_file_name, '', f'Операция \'{operations[str(vertex)]}\'' не соответствует вершине \'{vertex}\''')

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('-i', '--input', required=True, help='Имя входного файла')
    parser.add_argument('-o', '--output', help='Имя выходного файла')
    parser.add_argument('--operations', help='Имя файла с описанием операций')
    parser.add_argument('--log-file', help='Имя файла с логом программы',
dest='log_file')
    parser.add_argument('--log-level', help='Уровень логирования', dest='log_level',
default='debug')

```

```

args = parser.parse_args()

numeric_level = getattr(logging, args.log_level.upper(), None)
if not isinstance(numeric_level, int):
    raise ValueError('Invalid log level: %s' % args.log_level)

logging.basicConfig(level=numeric_level, filename=args.log_file, encoding='utf-
8')

try:
    all_edges, o_i_list = graph_reading_in_format(args.input)

    start = []
    for x in all_edges:
        if not len(all_edges[x]):
            start.append(x)

    cycle_validator(all_edges, o_i_list)

    for x in range(1, len(o_i_list) + 1):
        tmp = o_i_list[x][0]
        o_i_list[x][0] = o_i_list[x][1]
        o_i_list[x][1] = tmp
    reverse = get_reverse_graph(all_edges)
    graph = [[]] * (len(all_edges) + 1)
    k = 0
    list = []

    for x in sorted(reverse.keys()):
        for i in range(len(reverse[x])):
            list.append([reverse[x][i]])
        graph[k + 1] = list
        k += 1
        list = []

    new_graph = []

    for i in graph:
        tmp = []

        for j in range(len(i)):
            tmp.append([i[j][0][0], i[j][0][1]])

        tmp.sort(key=lambda x: x[0])
        new_graph.append(tmp)

    output, fun = cycle_finding(new_graph, o_i_list)

    fun = [y for y in fun if y != '']
    operations = operations_reading_in_format(all_edges, args.operations)
    new_fun = []
    fun_string = ''

    for x in fun:
        for char in x:
            tmp = char
            if char.isdigit():
                tmp = operations[str(char)]
            fun_string += str(tmp)
        new_fun.append(fun_string)
        fun_string = ''

```

```

check_operation_correctness(reverse, operations, args.operations)

values = do_eval_operation(reverse, start, operations)

result = ''
for i in range(len(start)):
    line = ''.join(fun[i])
    line += ' = '
    line += ''.join(new_fun[i])
    line += ' = '
    line += str(values[start[i]])
    result += line + '\n'

if args.output is not None:
    with open(args.output, 'w') as file:
        file.write(result)
else:
    print(result)

except InputException as e:
    logging.fatal('Ошибка в данных входного файла %s в строке %s', e.input_file,
e.line)
except DataException as e:
    logging.fatal('Ошибка в логике данных входного файла %s в строке %s. Текст
ошибки %s', e.input_file, e.line,
e.message)
except CycleException as e:
    logging.fatal('Существует цикл между вершинами %s и %s', e.v, e.u)
except Exception as e:
    logging.fatal('Неизвестная ошибка')
    logging.exception(e)

if __name__ == '__main__':
    main()

```