САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе № 3
по курсу «Алгоритмы и структуры данных»
Тема: Графы

Выполнила:
Олейник П.Д.
К3143

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Оглавление

Задачи

Задача № 2. Компоненты

Теперь вы решаете сделать так, чтобы в лабиринте не было мертвых зон, то есть чтобы из каждой клетки был доступен хотя бы один выход. Для этого вы находите связные компоненты соответствующего неориентированного графа и следите за тем, чтобы каждый компонент содержал выходную ячейку.

Дан неориентированный граф с n вершинами и m ребрами. Нужно посчитать количество компонент связности в нем.

- **Формат ввода / входного файла (input.txt).** Неориентированный граф с n вершинами и m ребрами по формату 1.
- **Ограничения на входные данные.** $1 \leq n \leq 10^3$, $0 \leq m \leq 10^3$.
- **Формат вывода / выходного файла (output.txt).** Выведите количество компонент связности.
- Ограничение по времени. 5 сек.
- Ограничение по памяти. 512 мб.
- Пример:

input	output
4 2 1 2 3 2	2

В этом графе есть два компонента связности: 1, 2, 3 и 4.

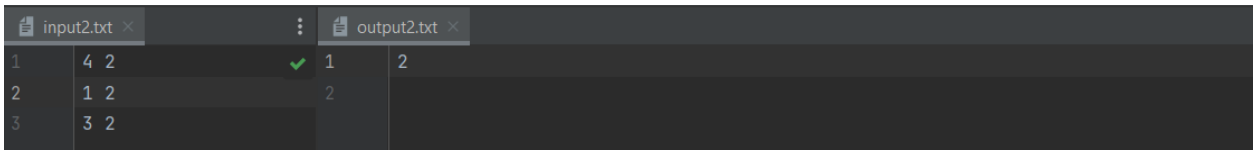
Листинг кода:

```
import time
import tracemalloc

t_start = time.perf_counter()
tracemalloc.start()


# nodes - список смежности
# visited - список посещенных вершин

def explore(nodes, visited, v):  # найти все вершины, достижимые из v
    visited[v] = True
    for w in nodes[v]:
        if not visited[w]:
            explore(nodes, visited, w)


def dfs(nodes):  # обход в глубину
    visited = [False for v in range(len(nodes))]
    cc = 1  # первая компонента связности
    for v in range(len(nodes)):
        if not visited[v]:
            explore(nodes, visited, v)
```

```
            cc += 1
    return cc - 1


def make_table_of_nodes(n, edges):
    nodes = [[] for v in range(n)]
    for i in range(len(edges)):
        v, w = [(int(i) - 1) for i in edges[i].split()]
        nodes[v].append(w)
        nodes[w].append(v)
    return nodes


def solve(n, edges):
    nodes = make_table_of_nodes(n, edges)
    return dfs(nodes)


def write_to_file():
    f = open("input2.txt")
    n, m = [int(i) for i in f.readline().split()]
    edges = [f.readline() for _ in range(m)]
    f.close()

    answer = solve(n, edges)

    file2 = open("output2.txt", "w+")
    file2.write(str(answer) + "\n")
    file2.close()


if __name__ == "__main__":
    write_to_file()

    print("Время выполнения: %s секунд " % (time.perf_counter() - t_start))
    print("Затраты памяти: %s КБ " % (tracemalloc.get_traced_memory()[0] / 2
** 10))
```

Текстовое объяснение решения.
nodes - список смежности, visited - список посещенных вершин. С Помощью функции explore выполняется поиск в глубину всех вершин смежной с данной. В функции dfs выполняется поиск в глубину по всем компонентам смежности, каждую компоненту отслеживает счетчик cc.

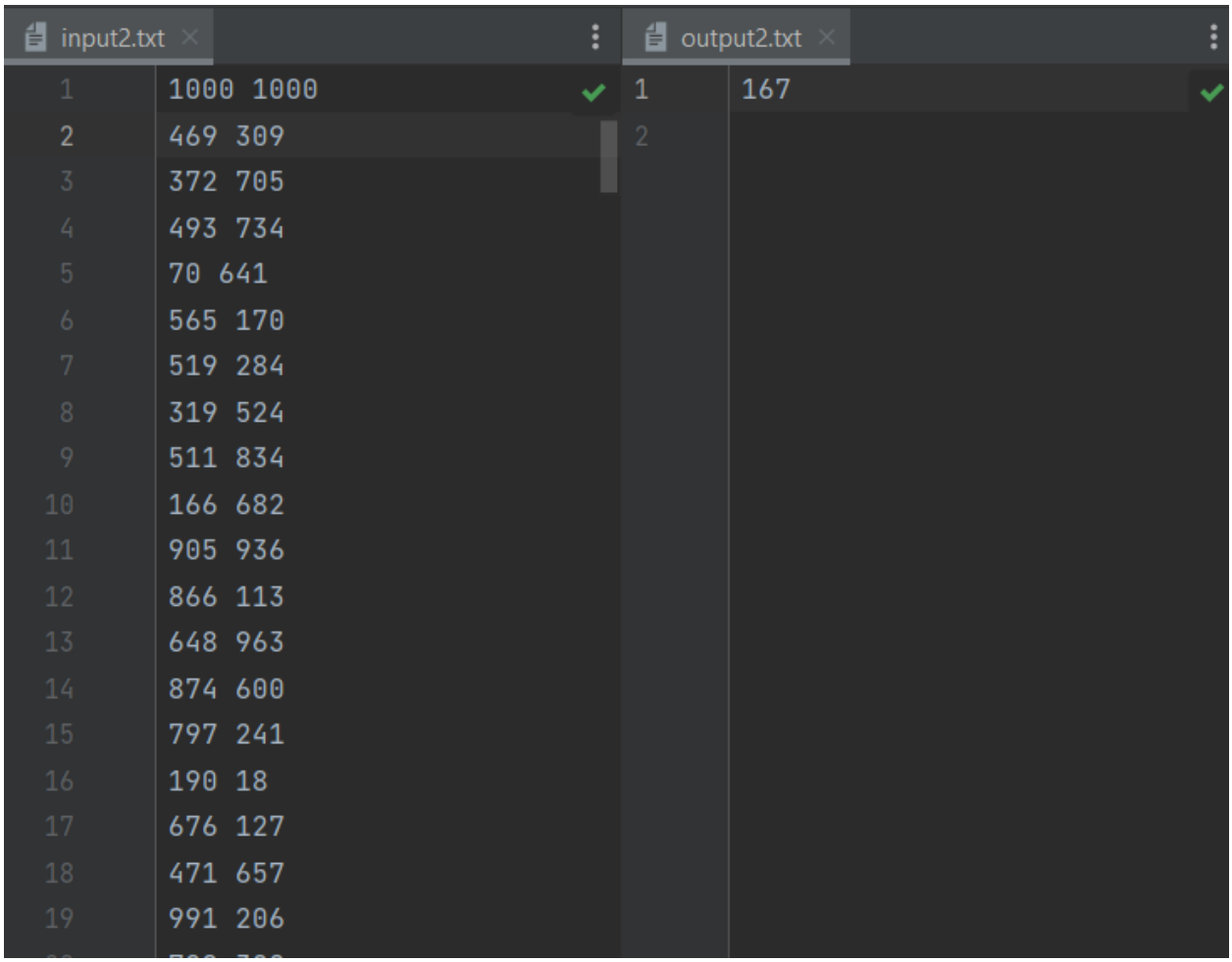
Результат работы кода на примерах из текста задачи:



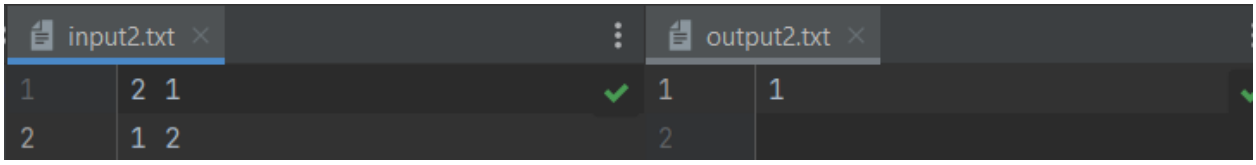
Время выполнения: 0.006764699995983392 секунд

Затраты памяти: 5.0849609375 КБ

Результат работы кода на максимальных и минимальных значениях:



Время выполнения: 0.02127130000735633 секунд



Время выполнения: 0.006009600008837879 секунд
Затраты памяти: 5.0302734375 КБ

Вывод по задаче: реализован поиск компонент связности неориентированного графа.

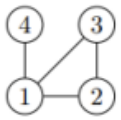
Задача №7. Двудольный граф

Неориентированный граф называется **двудольным**, если его вершины можно разбить на две части так, что каждое ребро графа соединяет вершины из разных частей, то есть не существует рёбер между вершинами одной и той же части графа. Двудольные графы естественным образом возникают в задачах, где граф используется для моделирования связей между объектами двух разных типов (например, мальчиками и девочками, или студентами и общежитиями). Альтернативное определение таково: граф двудольный, если его вершины можно раскрасить двумя цветами (например, черным и белым) так, что концы каждого ребра окрашены в разные цвета.

Дан неориентированный граф с n вершинами и m ребрами, проверьте, является ли он двудольным.

- **Формат ввода / входного файла (input.txt).** Неориентированный граф задан по формату 1.
- **Ограничения на входные данные.** $1 \le n \le 10^5, 0 \le m \le 10^5$.
- **Формат вывода / выходного файла (output.txt).** Выведите 1, если граф двудольный; и 0 в противном случае.
- Ограничение по времени. 10 сек.
- Ограничение по памяти. 512 мб.
- Пример 1:

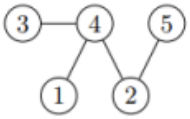
input	output
4 4 1 2 4 1 2 3 3 1	0



Этот граф не является двудольным. Чтобы убедиться в этом, предположим, что вершина 1 окрашена в белый цвет. Тогда вершины 2 и 3 нужно покрасить в черный цвет, так как граф содержит ребра 1, 2 и 1, 3. Но тогда ребро 2, 3 имеет оба конца одного цвета.

- Пример 2:

input	output
5 4 5 2 4 2 3 4 1 4	1



Этот граф двудольный: вершины 4 и 5 покрасим в белый цвет, все остальные вершины – в черный цвет.

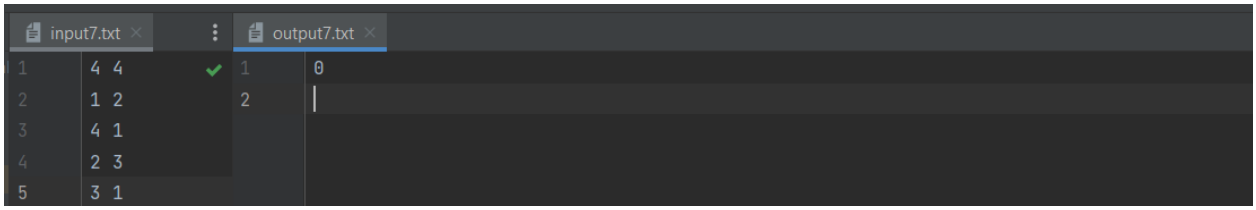
Листинг кода:

```
import time
import tracemalloc
from collections import deque

t_start = time.perf_counter()
tracemalloc.start()


def bfs(n, nodes, v):
    colors = [-1] * n
    colors[v] = 0
    queue = deque([v])
    while queue:
        u = queue.popleft()
        for v in nodes[u]:
            if colors[v] == colors[u]:
```

```
                return 0
            if colors[v] == -1:
                colors[v] = (colors[u] + 1) % 2
                queue.append(v)
    return 1


def make_table_of_nodes(n, edges):
    nodes = [[] for v in range(n)]
    for i in range(len(edges)):
        v, w = [(int(i) - 1) for i in edges[i].split()]
        nodes[v].append(w)
        nodes[w].append(v)
    return nodes


def solve(n, edges):
    nodes = make_table_of_nodes(n, edges)
    return bfs(n, nodes, 0)


def write_to_file():
    f = open("input7.txt")
    n, m = [int(i) for i in f.readline().split()]
    edges = [f.readline() for _ in range(m)]
    f.close()

    answer = solve(n, edges)

    file2 = open("output7.txt", "w+")
    file2.write(str(answer) + "\n")
    file2.close()


if __name__ == "__main__":
    write_to_file()

    print("Время выполнения: %s секунд " % (time.perf_counter() - t_start))
    print("Затраты памяти: %s КБ " % (tracemalloc.get_traced_memory()[0] / 2
** 10))
```

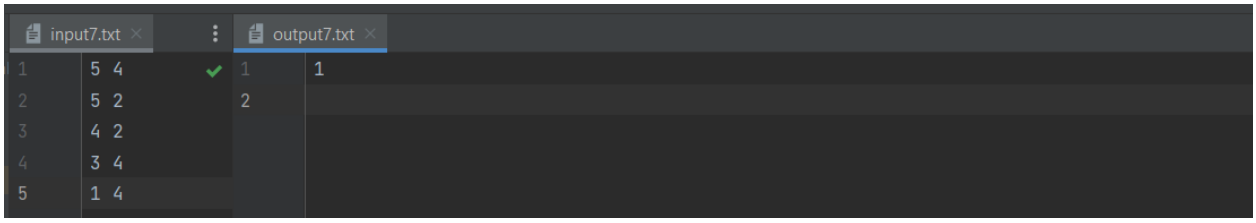
Текстовое объяснение решения.
Данная задача решается с помощью обхода в ширину. Каждый ярус “дерева”, получаемый при обходе раскрашивается в один из двух цветов. Если в какой-то момент две вершины из соседнего яруса оказались окрашены в один цвет, то граф не является двудольным. Во время обхода i-го яруса вершины, соединенные с вершинами i-го яруса и не посещенные ранее добавляются в очередь, которая содержит вершины яруса i+1. Пока очередь не пуста, обход продолжается.

Результат работы кода на примерах из текста задачи:



Время выполнения: 0.0012468999992111139 секунд

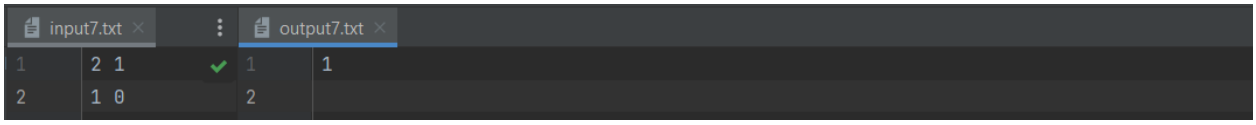
Затраты памяти: 5.0771484375 КБ



Время выполнения: 0.0036284999951021746 секунд

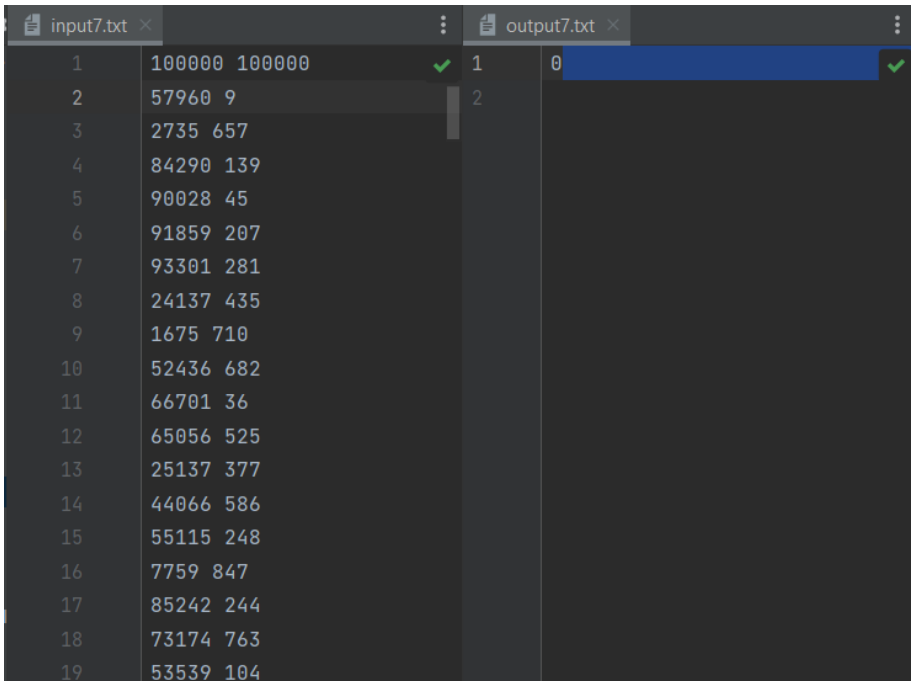
Затраты памяти: 4.5615234375 КБ

Результат работы кода на максимальных и минимальных значениях:



Время выполнения: 0.004211300009046681 секунд

Затраты памяти: 4.5615234375 КБ



Время выполнения: 1.1526821999868844 секунд

Затраты памяти: 13.0732421875 КБ

Вывод: реализована проверка графа на двудольность с помощью обхода в ширину.

Задача №16. Простейший неявный ключ

Рассмотрим программу, состоящую из n процедур $P_1, P_2, \ldots, P_n$. Пусть для каждой процедуры известны процедуры, которые она может вызывать. Процедура P называется потенциально рекурсивной, если существует такая последовательность процедур $Q_0, Q_1, \ldots, Q_k$, что $Q_0 = Q_k = P$ и для $i = 1...k$ процедура Q_{i-1} может вызвать процедуру Q_i. В этом случае задача будет заключаться в определении для каждой из заданных процедур, является ли она потенциально рекурсивной.

Требуется написать программу, которая позволит решить названную задачу.

- **Формат входных данных (input.txt) и ограничения.** Первая строка входного файла INPUT.TXT содержит целое число n – количество процедур в программе ($1 \le n \le 100$). Далее следуют n блоков, описывающих процедуры. После каждого блока следует строка, которая содержит 5 символов «*».

  Описание процедуры начинается со строки, содержащий ее идентификатор, состоящий только из маленьких букв английского алфавита и цифр. Идентификатор непуст, и его длина не превосходит 100 символов. Далее идет строка, содержащая число k ($k \le n$) – количество процедур, которые могут быть вызваны описываемой процедурой. Последующие k строк содержат идентификаторы этих процедур – по одному идентификатору на строке.

  Различные процедуры имеют различные идентификаторы. При этом ни одна процедура не может вызвать процедуру, которая не описана во входном файле.

- **Формат выходных данных (output.txt).** В выходной файл OUTPUT.TXT для каждой процедуры, присутствующей во входных данных, необходимо вывести слово YES, если она является потенциально рекурсивной, и слово NO – в противном случае, в том же порядке, в каком они перечислены во входных данных.

- Ограничение по времени. 1 сек.
- Ограничение по памяти. 16 мб.
- Пример:

input.txt	output.txt
3 p1 2 p1 p2 ***** p2 1 p1 ***** p3 1 p1 *****	YES YES NO

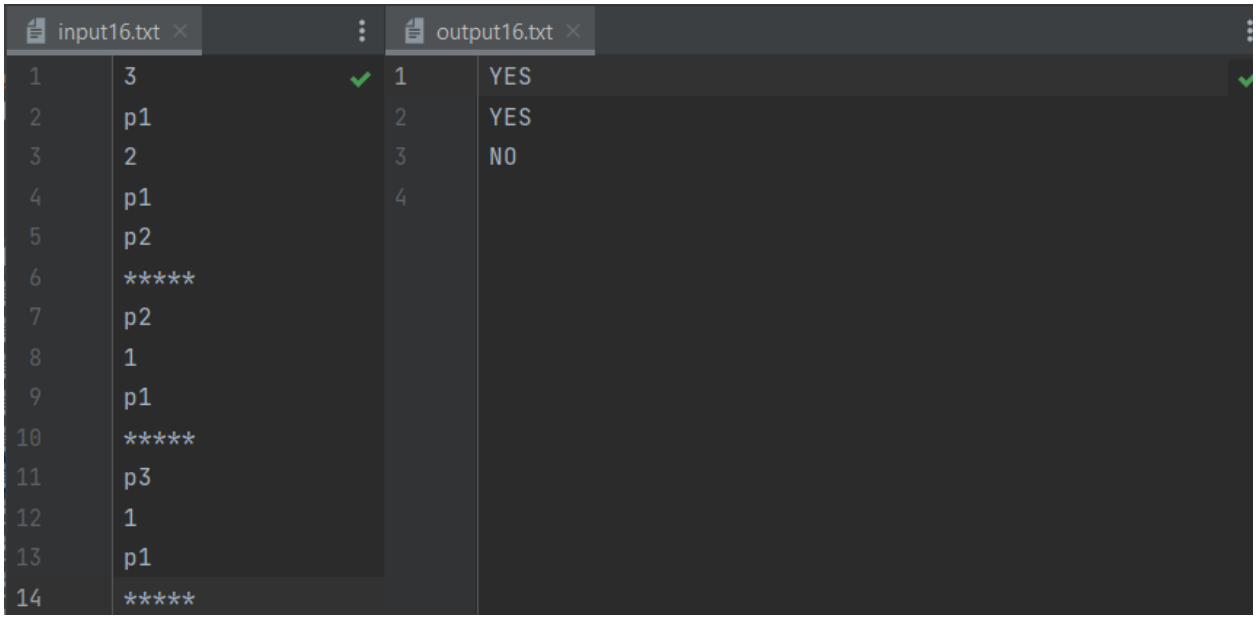
Листинг кода:

```
import time
import tracemalloc

t_start = time.perf_counter()
tracemalloc.start()


def explore(graph, visited, v):
    visited[v] = True
    for w in graph[v]:
        if not visited[w]:
            explore(graph, visited, w)


def find(graph, visited, v):
```

```
    for w in graph[v]:
        visited[w] = True
    for w in graph[v]:
        explore(graph, visited, w)


def check(graph, n, v):
    visited = [False for _ in range(n)]
    find(graph, visited, v)
    return 'YES' if visited[v] else 'NO'


def make_graph(n, processes):
    ids = dict()
    for i in range(n):
        ids[processes[i][0]] = i
    graph = []
    for i in range(n):
        graph.append([ids[pr] for pr in processes[i][1:]])
    return graph


def solve(n, processes):
    graph = make_graph(n, processes)
    return [check(graph, n, i) for i in range(n)]


def write_to_file():
    file1 = open("input16.txt")
    n = int(file1.readline())
    processes = []
    for i in range(n):
        pr = file1.readline()
        processes.append([pr])
        processes[-1].extend([file1.readline() for _ in
range(int(file1.readline()))])
        file1.readline()
    file1.close()

    answer = solve(n, processes)

    file2 = open("output16.txt", "w+")
    for el in answer:
        file2.write(el + "\n")
    file2.close()


if __name__ == "__main__":
    write_to_file()

    print("Время выполнения: %s секунд " % (time.perf_counter() - t_start))
    print("Затраты памяти: %s КБ " % (tracemalloc.get_traced_memory()[0] / 2
** 10))
```

Текстовое объяснение решения.

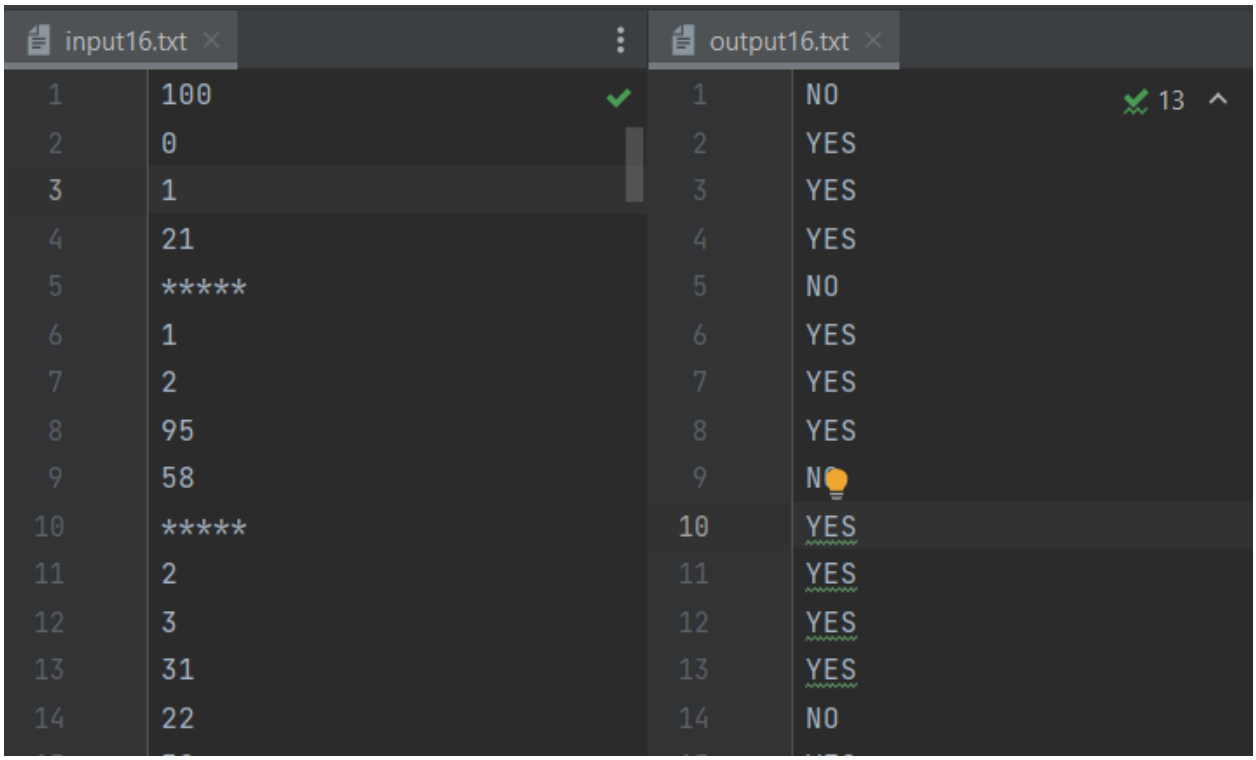
Результат работы кода на примерах из текста задачи:



Время выполнения: 0.0007302000012714416 секунд

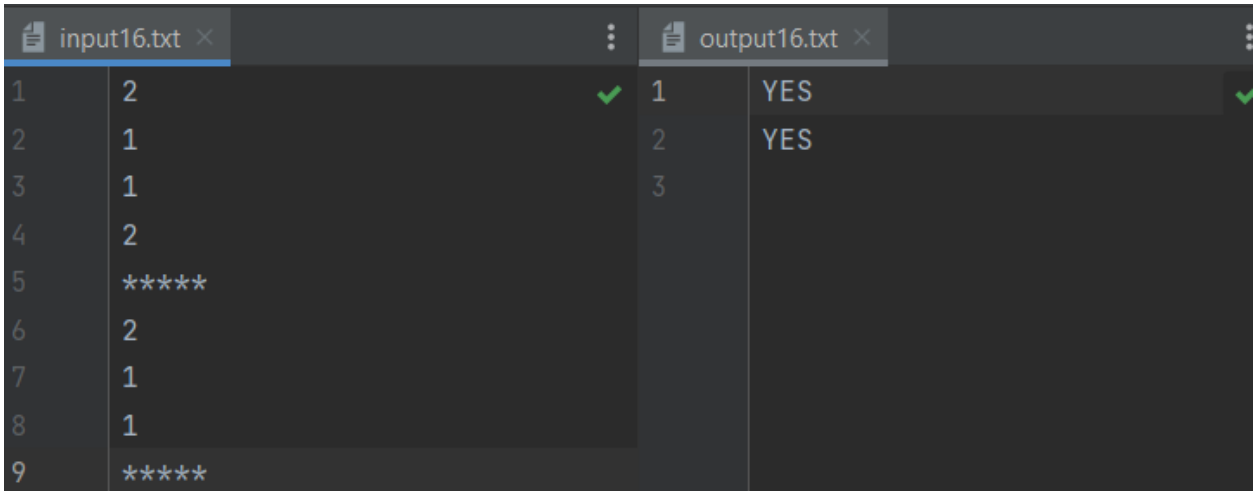
Затраты памяти: 6.1591796875 КБ

Результат работы кода на максимальных и минимальных значениях:



Время выполнения: 0.010936900012893602 секунд

Затраты памяти: 32.080078125 КБ



Время выполнения: 0.0009740999958012253 секунд

Затраты памяти: 6.1591796875 КБ

Задача	Язык	Результат	Тест	Время	Память
0345	Python	Accepted		0,125	1646 Кб

Вывод: данная задача решается с помощью обхода в глубину.

Вывод

Для хранения графов можно выбрать один из трех способов: список ребер (не самый эффективный вариант), с писок смежности (удобен для разряженных графов) и матрица смежности (подходит для плотных графов). Время работы алгоритмов га графах зависит не только от количества вершин, но и от количества ребер, т.е. определяется двумя переменными. На структуре графа удобен алгоритм поиска в глубину для топологической сортировки, а также для определения компонент связности. Поиск в ширину используется для нахождения длины кратчайшего пути в невзвешенном графе. Алгоритм Дейкстры (жадный алгоритм) позволяет найти кратчайший путь во взвешенном графе с неотрицательными весами.