Noor Nimrat Kaur
101903591
2CO23

Solution 1 :

```
class Pipe:
    def __init__(self, pipe_from, pipe_to, diameter):
        self.dia = diameter
        self.is_from_house = pipe_from
        self.goes_to_house = pipe_to
        self.color = None


class water_Problem:
    def __init__(self):
        self.taps = 0  # will be equal to the number of tap and tank pairs = num colonies
        self.color_token = 1  # for differentiation
        self.top_order = []  # topological ordering
        self.pipe_list = {}  # it's indexed by the house it starts from
        self.num_houses = 0
        self.num_pipes = 0
        self.output_list = []

    def get_input(self):
        #  get the number of houses and pipes
        [self.num_houses, self.num_pipes] = [int(i) for i in input().strip().split(" ")]

        for pipe in range(self.num_pipes):
            [is_from, goes_to, dia] = [int(i) for i in input().strip().split(" ")]

            new_pipe = Pipe(is_from, goes_to, dia)
            self.pipe_list[is_from] = new_pipe  # pipe associated with it

    def dfs(self, house):
        pipe = self.pipe_list[house]

        # self.print_pipe(house)

        if pipe.color is None:  # visited
            pipe.color = self.color_token  # 1 2 3 4

            if pipe.goes_to_house in self.pipe_list:  # checking if its end node
                [new_color, new_dia] = self.dfs(pipe.goes_to_house)

                pipe.color = new_color

                pipe.dia = min(pipe.dia, new_dia)  # because its a bottleneck
            else:
                self.taps += 1
                pipe.color = self.color_token
                # print("Num of taps = "+str(self.taps))
```

```
            self.top_order.append(pipe.goes_to_house)

        self.top_order.append(house)

    return [pipe.color, pipe.dia]

def topological_sort(self):
    for house in range(self.num_houses):
        if house in self.pipe_list and self.pipe_list[house].color is None:  # not an end node and not
visited
            self.dfs(house)
            self.color_token += 1
            # print("new color token =" + str(self.color_token))
    self.top_order.reverse()

def find_last(self, current_house):
    if current_house not in self.pipe_list:
        return current_house
    else:
        return self.find_last(self.pipe_list[current_house].goes_to_house)

def print_pipe(self, house):
    pipe = self.pipe_list[house]
    print(pipe.is_from_house, pipe.goes_to_house, pipe.dia, pipe.color)

def print_pipes(self):
    for pipe in self.pipe_list.keys():
        self.print_pipe(pipe)

def print_resulting_pipe(self, house):
    self.output_list.append([house, self.find_last(house), self.pipe_list[house].dia])

def output_1(self):
    print(self.taps)

def output_2(self):
    color_list = list(range(1, self.taps + 1))

    for house in self.top_order:
        if house in self.pipe_list:
            current_color = self.pipe_list[house].color
            if current_color in color_list:

                self.print_resulting_pipe(house)
                color_list.remove(current_color)
                if len(color_list) == 0:
                    break

    self.output_list.sort()  # Isn't necessary , done to match the output in the assignment
    for pipe in self.output_list:
        print(pipe[0], pipe[1], pipe[2])

def solve(self):
    self.topological_sort()
    print(self.top_order)
    self.output_1()
    self.output_2()
```

```
problem = water_Problem()
problem.get_input()
problem.solve()
```

Time Complexity : O(V+E)
Primary Algorithm : Topological sort

Solution 2:

```
from collections import deque
import numpy
# deque : a two ended queue

def get_graph():
    graph = []

    while True:
        row_in_matrix = input()

        if len(row_in_matrix) == 0:
            break
        row_in_matrix = [int(i) for i in row_in_matrix.strip().split()]
        graph.append(row_in_matrix)

    return graph


# print(get_graph())

class BFS:
    def __init__(self, graph):
        self.graph = graph
        self.cue = deque()
        self.rows = len(graph)
        self.cols = len(graph[0])
        self.visited = numpy.zeros((self.rows, self.cols), dtype=bool)

    def in_graph(self, node):
        row_index = node[0]
        col_index = node[1]
        if row_index >= self.rows or row_index < 0 or col_index >= self.cols or col_index < 0:
            return False
        return True

    def neighbours(self, current_node):
        row_index = current_node[0]
        col_index = current_node[1]

        up = [row_index - 1, col_index]
        down = [row_index + 1, col_index]
        left = [row_index, col_index - 1]
        right = [row_index, col_index + 1]
```

```
        return [up, right, down, left]

    def expand_node(self, current_node):

        for neighbour in self.neighbours(current_node):
            row = neighbour[0]
            col = neighbour[1]

            if self.in_graph(neighbour) and not self.visited[row][col]:
                self.cue.append(neighbour)
                self.visited[row][col] = True

    def bfs(self, start_index):

        self.cue.append(start_index)
        self.visited[start_index[0]][start_index[1]] = True

        while len(self.cue) != 0:
            current_node = self.cue.popleft()
            row = current_node[0]
            col = current_node[1]

            print(self.graph[row][col], end=" ")
            self.expand_node(current_node)


bfs_graph = get_graph()
problem = BFS(bfs_graph)
problem.bfs([2, 2])
```

Time Complexity : O(V+E)

Primary Algorithm :BFS