**1:**

from collections import defaultdict

# using default dict instead of dict to eleminate KeyError

# dict to save paths

higherland = defaultdict(list)

*def* bfs(*higherland*, *initial\_position*, *destination*):

    explored\_positions = []

    position\_queue = [[*initial\_position*]]

    if *initial\_position* == *destination*:

        print("0")

        return

    while position\_queue:

        # print("queue", position\_queue)

        path = position\_queue.pop(0)

        # print("path", path)

        current\_position = path[-1]

        # print("current", current\_position)

        # print("exploxed", explored\_positions)

        if current\_position not in explored\_positions:

            adjacent\_positions = *higherland*[current\_position]

            for a\_position in adjacent\_positions:

                new\_path = list(path)

                new\_path.append(a\_position)

                position\_queue.append(new\_path)

                if a\_position == *destination*:

                    print(len(new\_path) - 1)

                    return

            explored\_positions.append(current\_position)

    return

*def* connection(*u*, *v*):

    higherland[str(*u*)].append(str(*v*))

    higherland[str(*v*)].append(str(*u*))

if \_\_name\_\_ == '\_\_main\_\_':

    number\_of\_fixed\_positions = int(input())

    number\_of\_connections = int(input())

    for i in range(number\_of\_connections):

        edges = input().split()

        connection(edges[0], edges[1])

    linas\_position = input()

    # print(highland)

    bfs(higherland, '0', linas\_position)

**2:**

from collections import defaultdict

# using default dict instead of dict to eleminate keyError

# dict to save paths

highland = defaultdict(list)

*def* bfs(*highland*, *initial\_position*, *destination*):

    explored\_positions = []

    position\_queue = [[*initial\_position*]]

    if *initial\_position* == *destination*:

        print("0")

        return

    while position\_queue:

        # print("queue", position\_queue)

        path = position\_queue.pop(0)

        # print("path", path)

        current\_position = path[-1]

        # print("current", current\_position)

        # print("exploxed", explored\_positions)

        if current\_position not in explored\_positions:

            adjacent\_positions = *highland*[current\_position]

            for a\_position in adjacent\_positions:

                new\_path = list(path)

                new\_path.append(a\_position)

                position\_queue.append(new\_path)

                if a\_position == *destination*:

                    return print(len(new\_path) - 1)

            explored\_positions.append(current\_position)

    return

*def* connection(*u*, *v*):

    highland[str(*u*)].append(str(*v*))

    highland[str(*v*)].append(str(*u*))

if \_\_name\_\_ == '\_\_main\_\_':

    number\_of\_fixed\_positions = int(input())

    number\_of\_connections = int(input())

    for i in range(number\_of\_connections):

        edges = input().split()

        connection(edges[0], edges[1])

    linas\_position = input()

    noras\_position = input()

    laras\_position = input()

    if bfs(highland, laras\_position, linas\_position) < bfs(highland, noras\_position, linas\_position):

        print('Lara')

    elif bfs(highland, laras\_position, linas\_position) > bfs(highland, noras\_position, linas\_position):

        print('Nora')

    else:

        print("It'll end in a tie")

**3:**

# should have used text files before to reduce testing errors

# or maybe unit-testing, but i'm just learning the basics of unit-testing, hope to use them in later labs

*def* bfs(*highest\_land*, *predecessor*):

    position\_queue = []

    position\_queue.insert(len(position\_queue), *predecessor*)

    visit = [False]\*int(lines[0])

    visit[*predecessor*] = True

    parents = ['null']\*int(lines[0])

    while position\_queue:

        v = position\_queue.pop()

        for n in *highest\_land*[v]:

            if visit[n] == False:

                position\_queue.insert(len(position\_queue), n)

                visit[n] = True

                parents[n] = v

    return parents

*def* shortest\_path(*parent*, *destination*, *predecessor*):

    shortest\_path = [*destination*]

    node = *parent*[*destination*]

    while *predecessor* != node:

        shortest\_path.insert(len(shortest\_path), node)

        node = *parent*[node]

    return shortest\_path

shortest\_path\_list = []

if \_\_name\_\_ == '\_\_main\_\_':

    with open('Task\_03.txt') as f:

        lines = [line.rstrip() for line in f]

    total\_positions = int(lines[0])

    total\_connections = int(lines[1])+2

    highest\_land = [[] for i in range(total\_positions)]

    for i in range(2, total\_connections):

        u = int(lines[i].split(' ')[0])

        v = int(lines[i].split(' ')[1])

        highest\_land[v].append(u)

    find = (bfs(highest\_land, int(lines[int(lines[1])+2])))

    p = []

    for i in range(total\_connections+2, int(lines[total\_connections+1])+(total\_connections+2)):

        p.append(len(shortest\_path(find, int(lines[i]), int(lines[total\_connections]))))

    print(min(p))

**Task\_03.txt :**

10

14

0 1

0 2

0 3

1 3

1 4

2 3

3 5

4 7

4 8

5 6

6 7

6 9

7 8

8 9

9

5

0

1

3

5

7