Week - 4, Practice Programming

Problem 1

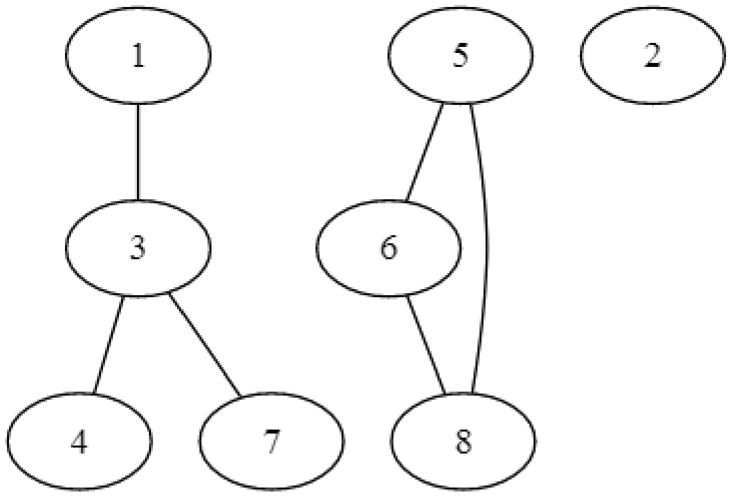
Given an undirected graph `G`, write a Python function to compute the number of connected components. A set of nodes form a connected component in an undirected graph if there exists a path between every pair of nodes in this set.

Write a Python function `findComponents_undirectedGraph(vertices, edges)`, that accepts a list of vertices and a list of tuples that represent edges, and returns the number of connected components in the graph formed by `vertices` and `edges`. Each tuple `(i,j)` in `edges` represents an edge between vertices `i` and `j`.

For a completely connected graph there is only one connected component, hence the function should return `1`

For the below graph, `G1`, the number of connected components is `3`. So the function should return `3`.

`G1`



Sample Input: `For graph G1`

```
1 2 3 4 5 6 7 8     # Vertices
6                               # Number of edges
1 3                          # edge
3 4                          # edge
3 7                          # edge
5 6                          # edge
5 8                          # edge
6 8                          # edge
```

Return:

```
3
```

Solution:

```python
from collections import deque
class myQueue:
  def __init__(self):
    self.Q = deque()

  def deQueue(self):
    return self.Q.popleft()

  def enQueue(self, x):
    return self.Q.append(x)

  def isEmpty(self):
    return False if self.Q else True

# Print number of connected components for undirected graph. This method will not work for directed graphs.
def findComponents_undirectedGraph(vertices, edges):
  # Create a adjacency list for graph.
  GList = {}
  for i in vertices:
    GList[i]=[]
  for (i,j) in edges:
    GList[i].append(j)
    GList[j].append(i)

  # Mark every vertex not visited.
  visited = {v:False for v in vertices}

  q = myQueue()
  componentsCount = 0

  # 1. Select some vertex v
  # 2. Start traversing the graph from v, till all vertices are visited in this component. Increment component count.
  # 3. Search for any unvisited vertex v, go to step 2
  for v in vertices:
    if not visited[v]:
      q.enQueue(v)

      while not q.isEmpty():
        v = q.deQueue()
        if not visited[v]:
          for i in GList[v]:
            if(not visited[i]):
              q.enQueue(i)
          visited[v]=True

      componentsCount += 1

  return componentsCount
```

Suffix

```python
v = [item for item in input().split(" ")]
numberOfEdges = int(input())
e = []
for i in range(numberOfEdges):
  s = input().split(" ")
  e.append((s[0], s[1]))
print(findComponents_undirectedGraph(v, e))
```

Public Test Case 1

Input

```
1 2 3 4 5 6 7 8
6
1 3
3 4
3 7
5 6
5 8
6 8
```

Output

```
3
```

Public Test Case 2

Input

```
a b c d e f g h i j
7
a c
c d
c g
e f
e h
f h
b i
```

Output

```
4
```

Private Test Case 1

Input

```
1 2 3 4 5 6 7 8
8
1 3
3 4
3 7
5 6
5 8
6 8
2 8
7 8
```

Output

```
1
```

Private Test Case 2

Input

```
1 2 3 4 5 6 7 8 9
6
1 3
3 4
3 7
5 6
5 8
6 8
```

Output

```
4
```

Private Test Case 3

Input

```
1 2 3 4 5 6 7 8
6
1 3
3 4
3 7
5 6
5 8
6 8
```

Output

```
3
```

Private Test Case 4

Input

```
a b c d e f g h i j k l m n o11a cc dc ge fe hf hb ij lj nk om o
```

Output

```
5
```

Private Test Case 5

Input

```
a b c d e f g h i j k l m n o
1
a c
```

Output

```
14
```

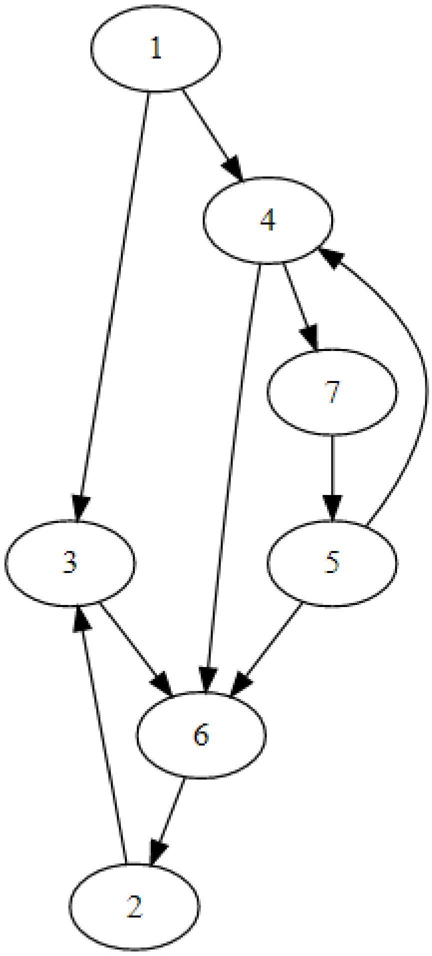
Problem 2

Complete the Python function `findAllPaths` to find all possible paths from the source vertex to destination vertex in a directed graph.

Function **findAllPaths(vertices, gList, source, destination)** takes `vertices` as a list of vertices, `gList` a dictionary that is an adjacency List representation of graph edges, `source` vertex, `destination` vertex, and returns a list of all paths from `source` to `destination`. The return value will be a List of Lists, where every path is a sequence of vertices as a List. Return an empty list if no path exists from 'source' to 'destination'.

```python
def findAllPaths(vertices, gList, source, destination):
  # Your function definition goes here.
```

For the graph below



Sample Input:

```
vertices: [1, 2, 3, 4, 5, 6, 7, 8],
gList: {1:[3,4], 2:[3], 3:[6], 4:[6,7], 5:[4,6], 6:[2], 7:[5]},
source: 1
destination: 2
```

Return:

```
[[1, 3, 6, 2],
 [1, 4, 6, 2],
 [1, 4, 7, 5, 6, 2]]
```

Solution:

```
# Helper functions
from collections import deque
class myQueue:
  def __init__(self):
    self.Q = deque()

  def deQueue(self):
    return self.Q.popleft()

  def enQueue(self, x):
    return self.Q.append(x)

  def isEmpty(self):
    return False if self.Q else True

# Function
def findAllPaths(vertices, gList, source , destination):
  allPaths=[]
  path=[]
  visited = {v:False for v in vertices}
  findAllPathsRecursive(vertices, gList, source, destination, visited, path, allPaths)
  return allPaths

# Function that will be called recursively to find path from original source to destination, that passes through vertex 'src'.
# If a path is found add it o allPaths.
def findAllPathsRecursive(vertices, gList, src, dest, visited, path, allPaths):
  visited[src] = True
  path.append(src)

  if (src == dest):
    allPaths.append(path.copy())

  for e in gList[src]:
    if not visited[e]:
      findAllPathsRecursive(vertices, gList, e, dest, visited, path, allPaths)

  # If no path exist passing through this vertex remove it from path.
  # Mark it unvisited, this vertex could be part of some other path.
  path.pop()
  visited[src]=False
```

Suffix

```
#Vertices are expected to be labelled as single letter or single digit

```

```
#Sort and arrange the result for uniformity
def ArrangeResult(result):
  res = []
  for item in result:
    s = ""
    for i in item:
      s += str(i)
    res.append(s)
  res.sort()
  return res

v = [item for item in input().split(" ")]
Alist = {}
for i in v:
  Alist[i] = [item for item in input().split(" ") if item != '']
source = input()
dest = input()
print(ArrangeResult(findAllPaths(v, Alist, source, dest)))
```

Public Test Case 1

Input

```
1 2 3 4 5 6 7
3 4
3
6
6 7
4 6
2
5
1
2
```

Output

```
['1362', '1462', '147562']
```

Public Test Case 2

Input

```
a b c d e f g h i j
c
i
d g

f h
h




b
d
```

Output

```
[]
```

Private Test Case 1

Input

```
1 2 3 4 5 6 7
3 4
3
6
6 7
1 6
2
5
4
6
```

Output

```
['46', '475136', '4756']
```

Private Test Case 2

Input

```
a b c d e f g h i
c
i
d g

f h
h



g
h
```

Output

```
[]
```

Private Test Case 3

Input

```
a b c d e f g h i
c e
i
d g

f h
h

d

a
d
```

Output

```
['acd', 'aefhd', 'aehd']
```

Private Test Case 4

Input

```
a b c d e f g h i
c e
i
d g

f h
h
e
d g

a
d
```

Output

```
['acd', 'acgefhd', 'acgehd', 'aefhd', 'aehd']
```

Private Test Case 5

Input

```
a b c d e f g
b c d e f g
a c d e f g
a b d e f g
a b c e f g
a b c d f g
a b c d e g
a b c d e f
c
g
```

Output

```
['cabdefg', 'cabdeg', 'cabdfeg', 'cabdfg', 'cabdg', 'cabedfg', 'cabedg',
'cabefdg', 'cabefg', 'cabeg', 'cabfdeg', 'cabfdg', 'cabfedg', 'cabfeg',
'cabfg', 'cabg', 'cadbefg', 'cadbeg', 'cadbfeg', 'cadbfg', 'cadbg',
'cadebfg', 'cadebg', 'cadefbg', 'cadefg', 'cadeg', 'cadfbeg', 'cadfbg',
'cadfebg', 'cadfeg', 'cadfg', 'cadg', 'caebdfg', 'caebdg', 'caebfdg',
'caebfg', 'caebg', 'caedbfg', 'caedbg', 'caedfbg', 'caedfg', 'caedg',
'caefbdg', 'caefbg', 'caefdbg', 'caefdg', 'caefg', 'caeg', 'cafbdeg',
'cafbdg', 'cafbedg', 'cafbeg', 'cafbg', 'cafdbeg', 'cafdbg', 'cafdebg',
'cafdeg', 'cafdg', 'cafebdg', 'cafebg', 'cafedbg', 'cafedg', 'cafeg', 'cafg',
'cag', 'cbadefg', 'cbadeg', 'cbadfeg', 'cbadfg', 'cbadg', 'cbaedfg',
'cbaedg', 'cbaefdg', 'cbaefg', 'cbaeg', 'cbafdeg', 'cbafdg', 'cbafedg',
'cbafeg', 'cbafg', 'cbag', 'cbdaefg', 'cbdaeg', 'cbdafeg', 'cbdafg', 'cbdag',
'cbdeafg', 'cbdeag', 'cbdefag', 'cbdefg', 'cbdeg', 'cbdfaeg', 'cbdfag',
'cbdfeag', 'cbdfeg', 'cbdfg', 'cbdg', 'cbeadfg', 'cbeadg', 'cbeafdg',
'cbeafg', 'cbeag', 'cbedafg', 'cbedag', 'cbedfag', 'cbedfg', 'cbedg',
'cbefadg', 'cbefag', 'cbefdag', 'cbefdg', 'cbefg', 'cbeg', 'cbfadeg',
'cbfadg', 'cbfaedg', 'cbfaeg', 'cbfag', 'cbfdaeg', 'cbfdag', 'cbfdeag',
'cbfdeg', 'cbfdg', 'cbfeadg', 'cbfeag', 'cbfedag', 'cbfedg', 'cbfeg', 'cbfg',
'cbg', 'cdabefg', 'cdabeg', 'cdabfeg', 'cdabfg', 'cdabg', 'cdaebfg',
'cdaebg', 'cdaefbg', 'cdaefg', 'cdaeg', 'cdafbeg', 'cdafbg', 'cdafebg',
'cdafeg', 'cdafg', 'cdag', 'cdbaefg', 'cdbaeg', 'cdbafeg', 'cdbafg', 'cdbag',
'cdbeafg', 'cdbeag', 'cdbefag', 'cdbefg', 'cdbeg', 'cdbfaeg', 'cdbfag',
'cdbfeag', 'cdbfeg', 'cdbfg', 'cdbg', 'cdeabfg', 'cdeabg', 'cdeafbg',
'cdeafg', 'cdeag', 'cdebafg', 'cdebag', 'cdebfag', 'cdebfg', 'cdebg',
'cdefabg', 'cdefag', 'cdefbag', 'cdefbg', 'cdefg', 'cdeg', 'cdfabeg',
'cdfabg', 'cdfaebg', 'cdfaeg', 'cdfag', 'cdfbaeg', 'cdfbag', 'cdfbeag',
'cdfbeg', 'cdfbg', 'cdfeabg', 'cdfeag', 'cdfebag', 'cdfebg', 'cdfeg', 'cdfg',
'cdg', 'ceabdfg', 'ceabdg', 'ceabfdg', 'ceabfg', 'ceabg', 'ceadbfg',
'ceadbg', 'ceadfbg', 'ceadfg', 'ceadg', 'ceafbdg', 'ceafbg', 'ceafdbg',
'ceafdg', 'ceafg', 'ceag', 'cebadfg', 'cebadg', 'cebafdg', 'cebafg', 'cebag',
'cebdafg', 'cebdag', 'cebdfag', 'cebdfg', 'cebdg', 'cebfadg', 'cebfag',
'cebfdag', 'cebfdg', 'cebfg', 'cebg', 'cedabfg', 'cedabg', 'cedafbg',
'cedafg', 'cedag', 'cedbafg', 'cedbag', 'cedbfag', 'cedbfg', 'cedbg',
'cedfabg', 'cedfag', 'cedfbag', 'cedfbg', 'cedfg', 'cedg', 'cefabdg',
'cefabg', 'cefadbg', 'cefadg', 'cefag', 'cefbadg', 'cefbag', 'cefbdag',
'cefbdg', 'cefbg', 'cefdabg', 'cefdag', 'cefdbag', 'cefdbg', 'cefdg', 'cefg',
'ceg', 'cfabdeg', 'cfabdg', 'cfabedg', 'cfabeg', 'cfabg', 'cfadbeg',
'cfadbg', 'cfadebg', 'cfadeg', 'cfadg', 'cfaebdg', 'cfaebg', 'cfaedbg',
'cfaedg', 'cfaeg', 'cfag', 'cfbadeg', 'cfbadg', 'cfbaedg', 'cfbaeg', 'cfbag',
'cfbdaeg', 'cfbdag', 'cfbdeag', 'cfbdeg', 'cfbdg', 'cfbeadg', 'cfbeag',
'cfbedag', 'cfbedg', 'cfbeg', 'cfbg', 'cfdabeg', 'cfdabg', 'cfdaebg',
'cfdaeg', 'cfdag', 'cfdbaeg', 'cfdbag', 'cfdbeag', 'cfdbeg', 'cfdbg',
'cfdeabg', 'cfdeag', 'cfdebag', 'cfdebg', 'cfdeg', 'cfdg', 'cfeabdg',
'cfeabg', 'cfeadbg', 'cfeadg', 'cfeag', 'cfebadg', 'cfebag', 'cfebdag',
'cfebdg', 'cfebg', 'cfedabg', 'cfedag', 'cfedbag', 'cfedbg', 'cfedg', 'cfeg',
'cfg', 'cg']
```

Problem 3

Maze solver

Alice wants to find the key in a maze and get out of it. The maze representation is given below, where `X` represents walls, *space* represents the allowed tiles Alice can walk on and `*` represents the tile that has the key.

- There is only one tile opening in the left-most vertical wall, where Alice is initially standing.
- Similarly there is only one tile opening in the right-most vertical wall, from which Alice has to exit.
- Alice can travel horizontally or vertically, but cannot travel diagonally. Moving to adjacent tile vertically or horizontally is counted as a step.

There are three possible outcomes, either you can exit the maze after getting the key, or the key is not reachable or the finish tile is not reachable.

- Print the minimum number of steps Alice requires to reach the finish tile traveling through tile having the key.
- If the key tile is not reachable then print `-1`.
- If the key tile is reachable but finish tile is not reachable then print `-2`.

Note: Input and printing are <u>required</u>

Sample Input

```
XXXXXXXXXXXXXX
   X    XXX  X
X  X    X X  X
X  X         X
X  XX  X XX  X
X  X  XX  X
X     XX XXXXX
X  X         X
X  X      *  X
XXXXXXXXXXXXXX

```

Sample Output

```
31
```

Solution:

```python
# Create a graph with every tile as a vertex, with an edge betwwen adjacent
# tiles if Alice can travel between those tiles.
def preprocessing(maze):
  m, n = len(maze), len(maze[0])
  S, E, K = None, None, None
  AList = {}
  for i in range(m):
    for j in range(n):
      AList[(i,j)] = []
      allowedTiles = [' ', '*']
      if maze[i][j] in allowedTiles:
```

```
        if i+1 < m and maze[i+1][j] in allowedTiles:
          AList[(i,j)].append((i+1, j))
        if 0 <= i-1 and maze[i-1][j] in allowedTiles:
          AList[(i,j)].append((i-1, j))
        if j+1 < n and maze[i][j+1] in allowedTiles:
          AList[(i,j)].append((i, j+1))
        if 0 <= j-1 and maze[i][j-1] in allowedTiles:
          AList[(i,j)].append((i, j-1))
        if j == 0: S = (i,j)
        if j == n-1: E = (i,j)
        if maze[i][j] == '*': K = (i,j)
  return AList, S, E, K

# Do a BFS maintaining level information to get the number of steps required.
def BFS(AList, x):
  visited = {k:False for k in AList.keys()}
  level = {k:None for k in AList.keys()}
  q = []

  visited[x] = True
  level[x] = 0
  q.append(x)
  while len(q) > 0:
    v = q.pop(0)
    visited[v] = True
    for i in AList[v]:
      if not visited[i]:
        q.append(i)
        if level[i] == None:
          level[i] = level[v] + 1
  from pprint import pprint
  return level

maze = []
line = input()
while line:
  maze.append(line)
  line = input()

AList, S, E, K = preprocessing(maze)
level = BFS(AList, S)
if level[K] == None:
  print(-1)
else:
  level2 = BFS(AList, K)
  if level2[E] == None:
    print(-2)
  else:
    print(level[K] + level2[E])
```

Test cases

Public Test case 1

Input

```
XXXXXXXXXXXXXX
   X    XXX  X
X  X    X X  X
X  X         X
X  XX  X XX  X
X  X  XX  X
X     XX XXXXX
X  X         X
X  X      *  X
XXXXXXXXXXXXXX


```

Output

```
31
```

Public Test case 2

Input

```
XXXXXXXXXXXXXX
   X    XXX  X
X  X    X X  X
X  X         X
X  XX  X XX  X
X  X  XX  X
X     XX XXXXX
X  X     X   X
X  X    X *  X
XXXXXXXXXXXXXX


```

Output

```
-1
```

Public Test case 3

Input

```
XXXXXXXXXXXXXX
   X    XXX  X
X  X    X X  X
X  X    X    X
X  XX  X XX  X
X  X  XX  X
X     XX XXXXX
X  X         X
X  X      *  X
XXXXXXXXXXXXXX
```

Output

```
-2
```

Private Test case 1

Input

```
XXXXXXXXXXXXXXXXX
X  X      X  X
X  X    X X  X  X
X  X         X  X
   XX  X X   X  X
X  X  XX *X X   X
X     XX  XX X  X
X  X    XX   X  X
X  X         X  X
XXXXXXXXXXXXXXXXX
```

Output

```
-2
```

Private Test case 2

Input

```
XXXXXXXXXXXXXXXXX
X  X      X  X
X  X    X X  X  X
X  X    X    X  X
   XX  X X   X  X
X  X  XX *X     X
X     XX  XX X  X
X  X    XX   X  X
X  X         X  X
XXXXXXXXXXXXXXXXX
```

Output

```
-1
```

Private Test case 3

Input

```
XXXXXXXXXXXXXX
X  X      X  X
X  X    X X  X
X  X    X    X
   XX  X XX  X
X  X  XX  X  *
X     XX XXXXX
X  X         X
X  X         X
XXXXXXXXXXXXXX


```

Output

```
24
```

Private Test case 4

Input

```
XXXXXXXXXXXXXXX  X      X  XX  X    X X  XX  X    X    X*  XX  X XX  XX  X
XX  XX     XX XXXXXX  X         XX  X         XXXXXXXXXXXXXXX
```

Output

```
24
```

Private Test case 5

Input

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X             XXXXXXXXXXXX       *      XXXXXXXXXXXXXX
X XXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXXXX
X XX      XXX X       XXXX XX        XX XX        XXXX
X XX XXXXX XX XX XXXXXX XX XX XXXXXXXXX XX XXXXXX XXXX
X XX XXXXX XX XX XXXXXX XX XX XXXXXXXXX XX XXXXXX XXXX
X XX XXXXX XX XX XXXXXX XX XX XXXXXXXXX XX XXXXXX XXXX
     XXXXX XX XX XXXXXX XX XX XXXXXXXXX XX XXXXXX XX
XXXX       XX XX XXXXXX XX XX        XX XX        XX X
XXXX XXXXXXXX XX XXXXXX XX XXXXXXXXX XX XX XXXXXX XX X
XXXX XXXXXXXX XX XXXXXX XX XXXXXXXXX XX XX XXXXXX XX X
XXXX XXXXXXXX XX XXXXXX XX XXXXXXXXX XX XX XXXXXX XX X
XXXX XXXXXXXX X       XXXX XX        XX XX XXXXXX XX X
XXXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX X
XXXXXXXXXXXXX              XXXXXXXXXXXX              X
```

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX


```

Output

```
105
```