

Week - 4, Practice Programming

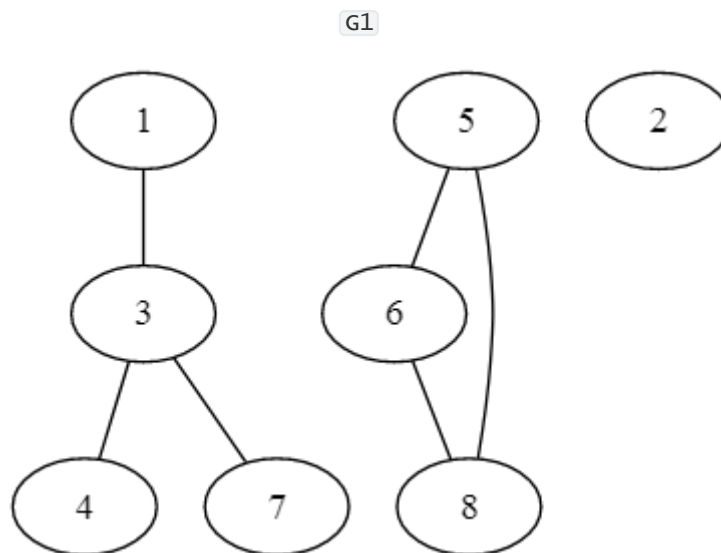
Problem 1

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`.



Sample Input: For graph `G1`

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

Return:

```
1 | 3
```

Solution:

```
1 from collections import deque
2 class myQueue:
3     def __init__(self):
4         self.Q = deque()
5
6     def dequeue(self):
7         return self.Q.popleft()
8
9     def enqueue(self, x):
10        return self.Q.append(x)
11
12    def isEmpty(self):
13        return False if self.Q else True
14
15    # Print number of connected components for undirected graph. This method
16    # will not work for directed graphs.
17    def findComponents_undirectedGraph(vertices, edges):
18        # Create a adjacency list for graph.
19        GList = {}
20        for i in vertices:
21            GList[i]=[]
22        for (i,j) in edges:
23            GList[i].append(j)
24            GList[j].append(i)
25
26        # Mark every vertex not visited.
27        visited = {v:False for v in vertices}
28
29        q = myQueue()
30        componentsCount = 0
31
32        # 1. Select some vertex v
33        # 2. Start traversing the graph from v, till all vertices are visited in
34        #    this component. Increment component count.
35        # 3. Search for any unvisited vertex v, go to step 2
36        for v in vertices:
37            if not visited[v]:
38                q.enqueue(v)
39
40            while not q.isEmpty():
41                v = q.dequeue()
42                if not visited[v]:
43                    for i in GList[v]:
44                        if(not visited[i]):
45                            q.enqueue(i)
46                    visited[v]=True
47
48            componentsCount += 1
49
50        return componentsCount
```

```
1 | v = [item for item in input().split(" ")]
2 | numberOfEdges = int(input())
3 | e = []
4 | for i in range(numberOfEdges):
5 |     s = input().split(" ")
6 |     e.append((s[0], s[1]))
7 | print(findComponents_undirectedGraph(v, e))
```

Public Test Case 1

Input

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

Output

```
1 | 3
```

Public Test Case 2

Input

```
1 | a b c d e f g h i j
2 | 7
3 | a c
4 | c d
5 | c g
6 | e f
7 | e h
8 | f h
9 | b i
```

Output

```
1 | 4
```

Private Test Case 1

Input

1	1 2 3 4 5 6 7 8
2	8
3	1 3
4	3 4
5	3 7
6	5 6
7	5 8
8	6 8
9	2 8
10	7 8

Output

1	1
---	---

Private Test Case 2

Input

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

Output

1	4
---	---

Private Test Case 3

Input

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

Output

1	3
---	---

Private Test Case 4

Input

```
1 | a b c d e f g h i j k l m n o
2 | 11
3 | a c
4 | c d
5 | c g
6 | e f
7 | e h
8 | f h
9 | b i
10 | j l
11 | j n
12 | k o
13 | m o
```

Output

```
1 | 5
```

Private Test Case 5

Input

```
1 | a b c d e f g h i j k l m n o
2 | 1
3 | a c
```

Output

```
1 | 14
```

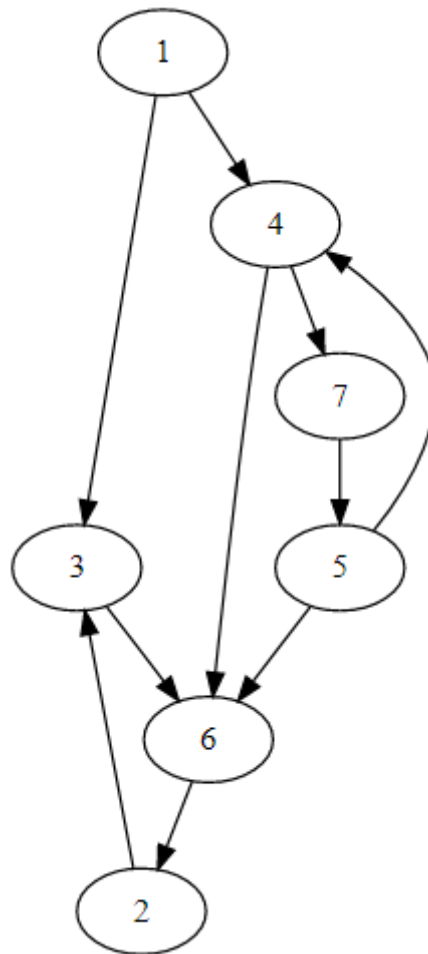
Problem 2

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

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

For the graph below



Sample Input:

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

Return:

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

Solution:

```
1 # Helper functions
2 from collections import deque
3 class myQueue:
4     def __init__(self):
5         self.Q = deque()
6
7     def dequeue(self):
8         return self.Q.popleft()
9
10    def enqueue(self, x):
11        return self.Q.append(x)
12
13    def isEmpty(self):
14        return False if self.Q else True
15
16 # Function
17 def findAllPaths(vertices, gList, source , destination):
18     allPaths=[]
19     path=[]
20     visited = {v:False for v in vertices}
21     findAllPathsRecursive(vertices, gList, source, destination, visited, path,
22 allPaths)
23     return allPaths
24
25 # Function that will be called recursively to find path from original source
26 # to destination, that passes through vertex 'src'.
27 # If a path is found add it o allPaths.
28 def findAllPathsRecursive(vertices, gList, src, dest, visited, path,
29 allPaths):
30     visited[src] = True
31     path.append(src)
32
33     if (src == dest):
34         allPaths.append(path.copy())
35
36     for e in gList[src]:
37         if not visited[e]:
38             findAllPathsRecursive(vertices, gList, e, dest, visited, path,
39 allPaths)
40
41     # If no path exist passing through this vertex remove it from path.
42     # Mark it unvisited, this vertex could be part of some other path.
43     path.pop()
44     visited[src]=False
```

Suffix

```
1 #Vertices are expected to be labelled as single letter or single digit
2
```

```

3  #Sort and arrange the result for uniformity
4  def ArrangeResult(result):
5      res = []
6      for item in result:
7          s = ""
8          for i in item:
9              s += str(i)
10         res.append(s)
11     res.sort()
12     return res
13
14 v = [item for item in input().split(" ")]
15 Alist = {}
16 for i in v:
17     Alist[i] = [item for item in input().split(" ") if item != '']
18 source = input()
19 dest = input()
20 print(ArrangeResult(findAllPaths(v, Alist, source, dest)))

```

Public Test Case 1

Input

```

1  1 2 3 4 5 6 7
2  3 4
3  3
4  6
5  6 7
6  4 6
7  2
8  5
9  1
10 2

```

Output

```

1  ['1362', '1462', '147562']

```

Public Test Case 2

Input

```

1  a b c d e f g h i j
2  c
3  i
4  d g
5
6  f h
7  h
8
9
10
11
12 b
13 d

```


Output

```
1 | []
```

Private Test Case 1

Input

```
1 | 1 2 3 4 5 6 7
2 | 3 4
3 | 3
4 | 6
5 | 6 7
6 | 1 6
7 | 2
8 | 5
9 | 4
10 | 6
```

Output

```
1 | ['46', '475136', '4756']
```

Private Test Case 2

Input

```
1 | a b c d e f g h i
2 | c
3 | i
4 | d g
5 |
6 | f h
7 | h
8 |
9 |
10 |
11 | g
12 | h
```

Output

```
1 | []
```

Private Test Case 3

Input

1	a b c d e f g h i
2	c e
3	i
4	d g
5	
6	f h
7	h
8	
9	d
10	
11	a
12	d

Output

1	['acd', 'aefhd', 'aehd']
---	--------------------------

Private Test Case 4

Input

1	a b c d e f g h i
2	c e
3	i
4	d g
5	
6	f h
7	h
8	e
9	d g
10	
11	a
12	d

Output

1	['acd', 'acgefhd', 'acgehd', 'aefhd', 'aehd']
---	-----------------------------------------------

Private Test Case 5

Input

1	a b c d e f g
2	b c d e f g
3	a c d e f g
4	a b d e f g
5	a b c e f g
6	a b c d f g
7	a b c d e g
8	a b c d e f
9	c
10	g

Output

1 ['cabdefg', 'cabdeg', 'cabdfeg', 'cabdfg', 'cabdg', 'cabedfg', 'cabedg',
'cabefdg', 'cabefg', 'cabeg', 'cabfdeg', 'cabfdg', 'cabfedg', 'cabfeg',
'cabfg', 'cabg', 'cadbefg', 'cadbeg', 'cadbfeg', 'cadbfg', 'cadbg',
'cadebfg', 'cadebg', 'cadebfg', 'cadefg', 'cadeg', 'cadfbeg', 'cadfbg',
'cadfebfg', 'cadfeg', 'cadfg', 'cadg', 'caebdfg', 'caebdg', 'caebfdg',
'caebfg', 'caebg', 'caedbfg', 'caedbg', 'caedfbg', 'caedfg', 'caedg',
'caefbdg', 'caefbg', 'caefdbg', 'caefdg', 'caefg', 'caeg', 'cafbdeg',
'cafbdg', 'cafbdeg', 'cafbeg', 'cafbg', 'cafdbeg', 'cafdbg', 'cafedbg',
'cafdeg', 'cafdg', 'cafebdg', 'cafebg', 'cafedbg', 'cafedg', 'cafeg', 'cafg',
'cag', 'cbadefg', 'cbadeg', 'cbadefg', 'cbadfg', 'cbadg', 'cbaedfg',
'cbaedg', 'cbaefdg', 'cbaefg', 'cbaeg', 'cbafdeg', 'cbafdg', 'cbafedg',
'cbafeg', 'cbafg', 'cbag', 'cbdaefg', 'cbdaeg', 'cbdafeg', 'cbdafg', 'cbdag',
'cbdea fg', 'cbdeag', 'cbdefag', 'cbdefg', 'cbdeg', 'cbdfaeg', 'cbdfag',
'cbdfeg', 'cbdfeg', 'cbdfg', 'cbdg', 'cbeadfg', 'cbeadg', 'cbeafd g',
'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', 'cdafebfg',
'cdafe g', 'cda fg', 'cdag', 'cdbaefg', 'cdbaeg', 'cdbafeg', 'cdbafg', 'cdbag',
'cdbeafg', 'cdbeag', 'cdbefag', 'cdbefg', 'cdbeg', 'cdbfaeg', 'cdbfag',
'cdbfeag', 'cdbfeg', 'cdbfg', 'cdbg', 'cdeabfg', 'cdeabg', 'cdeafbg',
'cdeafg', 'cdeag', 'cdebafg', 'cdebag', 'cdebfg', 'cdebfg', 'cdeb g',
'cdefabg', 'cdefag', 'cdefbag', 'cdefbg', 'cdefg', 'cdeg', 'cdfabeg',
'cdfabg', 'cdfaebg', 'cdfaeg', 'cdfag', 'cdfbaeg', 'cdfbag', 'cdfbeag',
'cdfbeg', 'cdfbg', 'cdfaebg', 'cdfaeg', 'cdfebag', 'cdfebg', 'cdfeg', 'cdfg',
'cdg', 'ceabdfg', 'ceabdg', 'ceabfdg', 'ceabfg', 'ceabg', 'ceadbfg',
'ceadb g', 'ceadfbg', 'ceadfg', 'ceadg', 'ceafbdg', 'ceafbg', 'ceafdbg',
'ceafdg', 'ceafg', 'ceag', 'cebadfg', 'cebadg', 'ceba fdg', 'ceba fg', 'cebag',
'cebdafg', 'cebdag', 'cebd fag', 'cebd fg', 'cebdg', 'cebfadg', 'cebfag',
'cebfdag', 'cebfdg', 'cebf g', 'cebg', 'cedabfg', 'cedabg', 'cedafbg',
'ceda fg', 'cedag', 'cedba fg', 'cedbag', 'cedbfag', 'cedbfg', 'cedbg',
'cedfabg', 'cedfag', 'cedfbag', 'cedfbg', 'cedfg', 'cedg', 'cefabdg',
'cefabg', 'cefa dbg', 'cefa dg', 'cefa g', '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 required

Sample Input

```
1  xxxxxxxxxxxxxx
2      x   xxx  x
3  x  x   x  x  x
4  x  x           x
5  x  xx  x  xx  x
6  x  x  xx  x
7  x      xx  xxxxx
8  x  x           x
9  x  x           *  x
10 xxxxxxxxxxxxxx
11
```

Sample Output

```
1  31
```

Solution:

```
1  # Create a graph with every tile as a vertex, with an edge between adjacent
2  # tiles if Alice can travel between those tiles.
3  def preprocessing(maze):
4      m, n = len(maze), len(maze[0])
5      S, E, K = None, None, None
6      AList = {}
7      for i in range(m):
8          for j in range(n):
9              AList[(i,j)] = []
10             allowedTiles = [' ', '*']
11             if maze[i][j] in allowedTiles:
```

```

12         if i+1 < m and maze[i+1][j] in allowedTiles:
13             AList[(i,j)].append((i+1, j))
14         if 0 <= i-1 and maze[i-1][j] in allowedTiles:
15             AList[(i,j)].append((i-1, j))
16         if j+1 < n and maze[i][j+1] in allowedTiles:
17             AList[(i,j)].append((i, j+1))
18         if 0 <= j-1 and maze[i][j-1] in allowedTiles:
19             AList[(i,j)].append((i, j-1))
20         if j == 0: S = (i,j)
21         if j == n-1: E = (i,j)
22         if maze[i][j] == '*': K = (i,j)
23     return AList, S, E, K
24
25     # Do a BFS maintaining level information to get the number of steps
    required.
26     def BFS(AList, x):
27         visited = {k:False for k in AList.keys()}
28         level = {k:None for k in AList.keys()}
29         q = []
30
31         visited[x] = True
32         level[x] = 0
33         q.append(x)
34         while len(q) > 0:
35             v = q.pop(0)
36             visited[v] = True
37             for i in AList[v]:
38                 if not visited[i]:
39                     q.append(i)
40                     if level[i] == None:
41                         level[i] = level[v] + 1
42         from pprint import pprint
43         return level
44
45     maze = []
46     line = input()
47     while line:
48         maze.append(line)
49         line = input()
50
51     AList, S, E, K = preprocessing(maze)
52     level = BFS(AList, S)
53     if level[K] == None:
54         print(-1)
55     else:
56         level2 = BFS(AList, K)
57         if level2[E] == None:
58             print(-2)
59         else:
60             print(level[K] + level2[E])

```

Test cases

Public Test case 1

Input

1	XXXXXXXXXXXXXXXXX
2	X XXX X
3	X X X X X
4	X X X
5	X XX X XX X
6	X X XX X
7	X XX XXXXX
8	X X X
9	X X * X
10	XXXXXXXXXXXXXXXXX
11	
12	

Output

1	31
---	----

Public Test case 2

Input

1	XXXXXXXXXXXXXXXXX
2	X XXX X
3	X X X X X
4	X X X
5	X XX X XX X
6	X X XX X
7	X XX XXXXX
8	X X X X
9	X X X * X
10	XXXXXXXXXXXXXXXXX
11	
12	

Output

1	-1
---	----

Public Test case 3

Input

```
1 |XXXXXXXXXXXXXXXXX
2 |  X   XXX  X
3 |X  X   X  X  X
4 |X  X   X   X
5 |X  XX  X  XX  X
6 |X  X  XX  X
7 |X      XX XXXXX
8 |X  X           X
9 |X  X          *  X
10|XXXXXXXXXXXXXXXXX
11|
12|
```

Output

```
1 | -2
```

Private Test case 1

Input

```
1 |XXXXXXXXXXXXXXXXX
2 |X  X      X  X
3 |X  X   X  X  X  X
4 |X  X           X  X
5 |   XX  X  X   X  X
6 |X  X  XX *X  X  X
7 |X      XX  XX  X  X
8 |X  X   XX   X  X
9 |X  X           X  X
10|XXXXXXXXXXXXXXXXX
11|
12|
```

Output

```
1 | -2
```

Private Test case 2

Input

```
1 |XXXXXXXXXXXXXXXXX
2 |X  X      X  X
3 |X  X   X  X  X  X
4 |X  X   X   X  X
5 |   XX  X  X   X  X
6 |X  X  XX *X   X
7 |X      XX  XX  X  X
8 |X  X   XX   X  X
9 |X  X           X  X
10|XXXXXXXXXXXXXXXXX
11|
12|
```

Output

1 | -1

Private Test case 3

Input

1 | XXXXXXXXXXXXXXXX
2 | X X X X
3 | X X X X X
4 | X X X X
5 | XX X XX X
6 | X X XX X *
7 | X XX XXXXX
8 | X X X
9 | X X X
10 | XXXXXXXXXXXXXXXX
11 |
12 |

Output

1 | 24

Private Test case 4

Input

1 | XXXXXXXXXXXXXXXX X X XX X X X XX X X X* XX X XX XX X
XX XX XX XXXXXX X XX X XXXXXXXXXXXXXXXX

Output

1 | 24

Private Test case 5

Input

1 | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
2 | X XXXXXXXXXXXXXXXX * XXXXXXXXXXXXXXXX
3 | X XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX
4 | X XX XXX X XXXX XX XX XX XXXX
5 | X XX XXXXX XX XX XXXXXX XX XX XXXXXXXXXXXXXXXX XX XXXXXX XXXX
6 | X XX XXXXX XX XX XXXXXX XX XX XXXXXXXXXXXXXXXX XX XXXXXX XXXX
7 | X XX XXXXX XX XX XXXXXX XX XX XXXXXXXXXXXXXXXX XX XXXXXX XXXX
8 | XXXXX XX XX XXXXXX XX XX XXXXXXXXXXXXXXXX XX XXXXXX XX
9 | XXXX XX XX XXXXXX XX XX XX XX XX X
10 | XXXX XXXXXXXXXXXXXXXX XX XXXXXX XX XXXXXXXXXXXXXXXX XX XX XXXXXX XX X
11 | XXXX XXXXXXXXXXXXXXXX XX XXXXXX XX XXXXXXXXXXXXXXXX XX XX XXXXXX XX X
12 | XXXX XXXXXXXXXXXXXXXX XX XXXXXX XX XXXXXXXXXXXXXXXX XX XX XXXXXX XX X
13 | XXXX XXXXXXXXXXXXXXXX XXXX XX XX XX XXXXXX XX X
14 | XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX X
15 | XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX X


```
16 | xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
17 |
18 |
```

Output

```
1 | 105
```

