

A few instructions

- Codes should be compatible with *Python3* and should run on Ubuntu.
- Code for each question should be placed in a separate stand-alone files.
- Codes should be well-commented.

1. Create a class `UndirectedGraph` that represents an *undirected graph* such that

[40]

- The graph should be stored in its adjacency list representation.
- It should be possible to create an undirected graph with a fixed number of vertices/nodes indexed as  $\{1, 2, \dots, n\}$  or a free graph (without any pre-defined number of vertices).

---

```
# The following code should create a free graph
g = UndirectedGraph()
```

---

---

```
# The following code should create a graph with 10 vertices
g = UndirectedGraph(10)
```

---

- It should be possible to add a node to the graph. If the graph is not free, an exception should be raised if index (only a positive integer should be allowed) of the node is larger than number of nodes in the graph.

---

```
g = UndirectedGraph()
g.addNode(1)
g.addNode(100)
```

---

---

```
g = UndirectedGraph(10)
g.addNode(4)
```

---

---

```
g = UndirectedGraph(10)
g.addNode(11)
```

---

Expected output:

```
<class 'Exception'>
Node index cannot exceed number of nodes
```

- It should be possible to add undirected edges to the graph.

**NOTE:** Adding an edge  $(a,b)$  to the graph implies that nodes  $a$  and  $b$  are also added to the graph.

```
# The following code adds an undirected edge (10,25) to the graph
g = UndirectedGraph()
g.addEdge(10, 25)
```

```
# The following code adds an undirected edge (1,2) to the graph
g = UndirectedGraph(10)
g.addEdge(1,2)
```

- Overload the  $+$  operator so that it will be possible to add nodes and edges to the graph using this operator.

```
# The following code adds node 10 to the graph
g = UndirectedGraph()
g = g + 10
```

```
# The following code adds an undirected edge (12,15) to the graph
g = UndirectedGraph()
g = g + (12, 15)
```

```
# The following code adds undirected edges (1,2) and (3,4) to the graph
g = UndirectedGraph(10)
g = g + (1, 2)
g = g + (3, 4)
```

- It should be possible to print an object of type `UndirectedGraph`.

```
g = UndirectedGraph()
print(g)
```

Expected output:

```
Graph with 0 nodes and 0 edges. Neighbours of the nodes are belows:
```

```
g = UndirectedGraph(5)
print(g)
```

Expected output:

```
Graph with 5 nodes and 0 edges. Neighbours of the nodes are belows:  
Node 1: {}  
Node 2: {}  
Node 3: {}  
Node 4: {}  
Node 5: {}
```

```
g = UndirectedGraph()  
g = g + 10  
g = g + (11, 12)  
print(g)
```

Expected output:

```
Graph with 3 nodes and 1 edges. Neighbours of the nodes are belows:  
Node 10: {}  
Node 11: {12}  
Node 12: {11}
```

```
g = UndirectedGraph(5)  
g = g + (1, 2)  
g = g + (3, 4)  
g = g + (1, 4)  
print(g)
```

Expected output:

```
Graph with 5 nodes and 3 edges. Neighbours of the nodes are belows:  
Node 1: {2, 4}  
Node 2: {1}  
Node 3: {4}  
Node 4: {1, 3}  
Node 5: {}
```

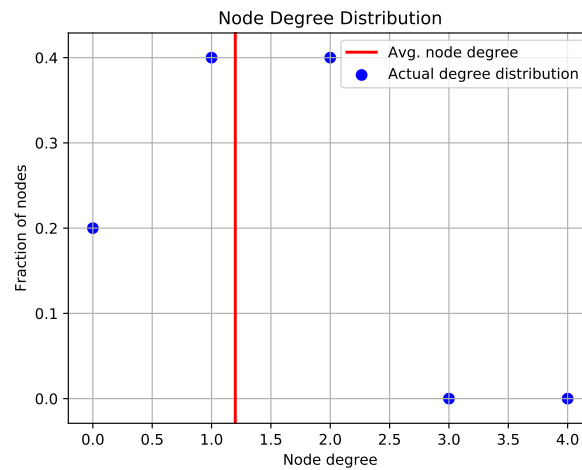
- It should be possible to plot the degree distribution<sup>1</sup> of a graph.

```
g = UndirectedGraph(5)  
g = g + (1, 2)  
g = g + (3, 4)  
g = g + (1, 4)  
g.plotDegDist()
```

---

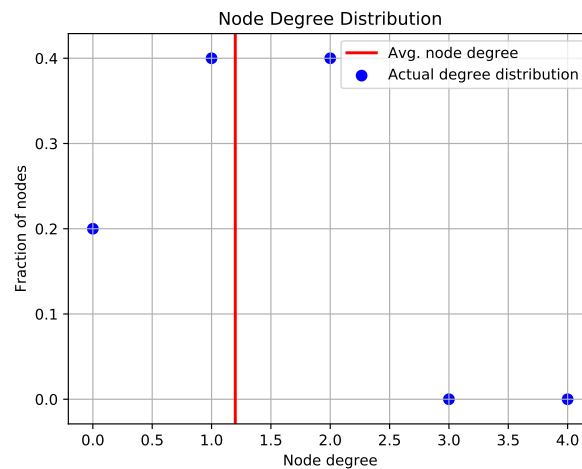
<sup>1</sup>[https://en.wikipedia.org/wiki/Degree\\_distribution](https://en.wikipedia.org/wiki/Degree_distribution)

Expected output:



```
g = UndirectedGraph()  
g = g + 100  
g = g + (1, 2)  
g = g + (1, 100)  
g = g + (100, 3)  
g = g + 20
```

Expected output:



2. Create a derived class **ERRandomGraph** from the base class **UndirectedGraph** such that it should be possible to create a Erdős-Rényi random graph  $G(n, p)$

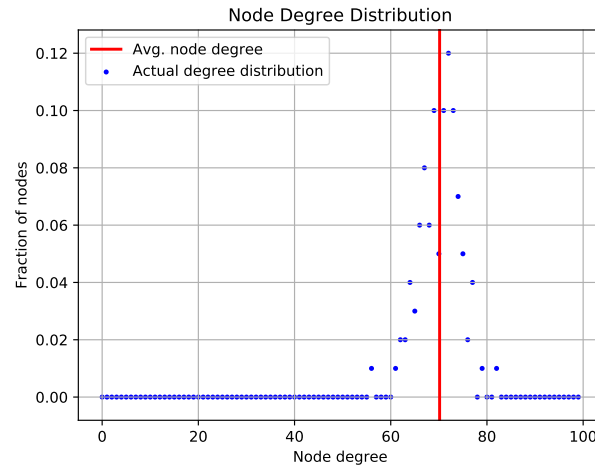
[20]

---

```
# The following code creates a G(100, 0.7) random graph and
# plots its degree distribution
g = ERRandomGraph(100)
g.sample(0.7)
g.plotDegDist()
```

---

Expected output:

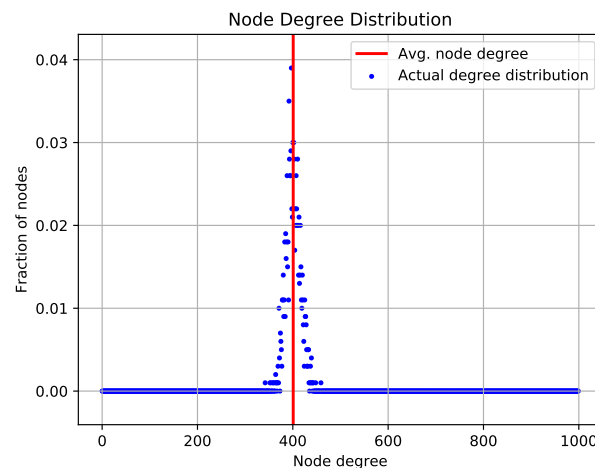



---

```
# The following code creates a G(1000, 0.4) random graph and
# plots its degree distribution
g = ERRandomGraph(1000)
g.sample(0.4)
g.plotDegDist()
```

---

Expected output:



3. *Connectivity* is a basic concept in Graph Theory. It defines whether a graph is *connected* or *disconnected*. A graph with fixed number of vertices is *connected* if there is a path between every pair of vertices.

[20]

- Add a method `isConnected` to the class `UndirectedGraph` that returns `True` if the graph is connected, and `False` otherwise.

**NOTE: This method should use BFS to determine connectedness and should not use any Python libraries/modules.**

---

```
g = UndirectedGraph(5)
g = g + (1, 2)
g = g + (2, 3)
g = g + (3, 4)
g = g + (3, 5)
print(g.isConnected())
```

---

Expected output:

True
------

---

```
g = UndirectedGraph(5)
g = g + (1, 2)
g = g + (2, 3)
g = g + (3, 5)
print(g.isConnected())
print(g)
```

---

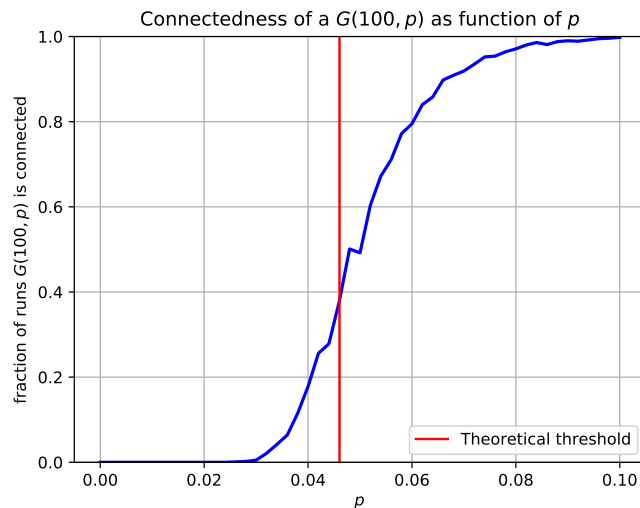
Expected output:

False
-------

- Write a function that uses the `isConnected` method of the `ERRandomGraph` class to verify the following statement

*“Erdős-Rényi random graph  $G(100, p)$  is almost surely connected only if  $p > \frac{\ln 100}{100}$ .”*

Expected output:



**NOTE:** The above plot shows connectedness averaged over 1000 runs.

4. A *connected component* of an undirected graph is a subgraph in which each pair of nodes is connected with each other via a path<sup>2</sup>.

[20]

- Add a method `oneTwoComponentSizes` to the class `UndirectedGraph` that returns size of largest and second largest connected components in the graph.

**NOTE:** This method should use BFS and should not use any Python libraries.

---

```
g = UndirectedGraph(6)
g = g + (1, 2)
g = g + (3, 4)
g = g + (6, 4)
print(g.oneTwoComponentSizes())
```

---

Expected output:

[3, 2]

---

```
g = RandomGraph(100)
g.sample(0.01)
print(g.oneTwoComponentSizes())
```

---

Expected output:

[23, 6]

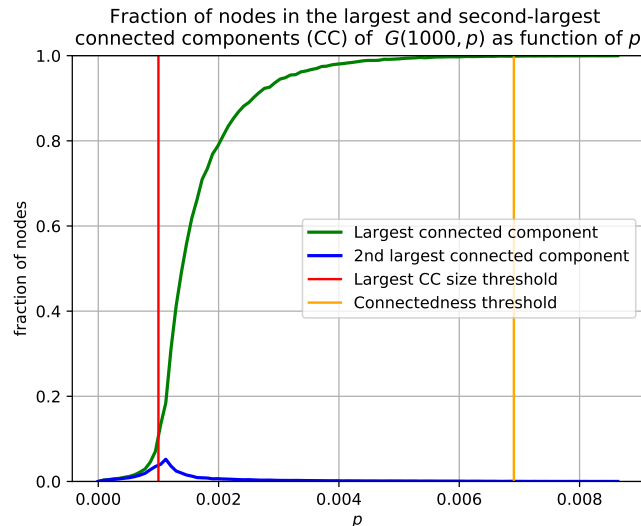
---

<sup>2</sup><https://www.baeldung.com/cs/graph-connected-components>

- Write a function that uses the `oneTwoComponentSizes` method of the `ERRandomGraph` class to verify the following statements

*“If  $p < 0.001$ , the Erdős-Rényi random graph  $G(1000, p)$  will almost surely have only small connected components. On the other hand, if  $p > 0.001$ , almost surely, there will be a single giant component containing a positive fraction of the vertices.”*

Expected output:



**NOTE:** The above plot shows size of largest and second-largest components averaged over 50 runs.

5. You are expected to use Python’s NetworkX package to solve this problem. Create a class `Lattice` such that

[30]

- It should be possible to create a  $n \times n$  grid graph (a lattice).

---

```
# The following code will create a 10 x 10 grid graph
l = Lattice(10)
```

---

- It should be possible to display the grid graph.

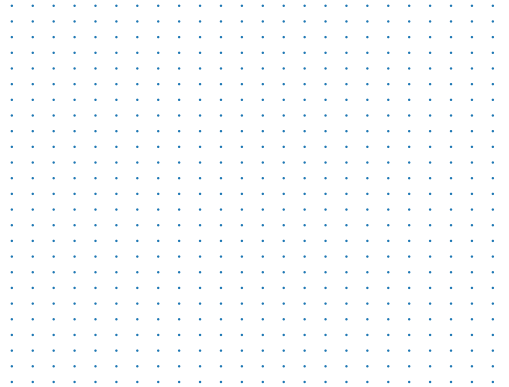
---

```
l = Lattice(25)
l.show()
```

---



Expected output:



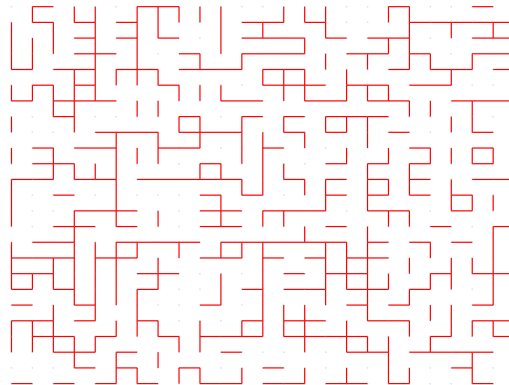
- It should be possible to simulate bond percolation<sup>3</sup> on the grid graph.

---

```
l = Lattice(25)
l.percolate(0.4)
l.show()
```

---

Expected output:



- The class should have a method `existsTopDownPath` that should return `True` if a path exists (along the open bonds) from the top-most layer to the bottom-most layer of the grid graph. The method should return `False` otherwise.

---

```
l = Lattice(25)
l.percolate(0.4)
l.existsTopDownPath()
```

---

- The class should have a method `showPaths` that, for every node  $u$  in the top-most layer, does either of the following

---

<sup>3</sup>[https://en.wikipedia.org/wiki/Percolation\\_theory](https://en.wikipedia.org/wiki/Percolation_theory)

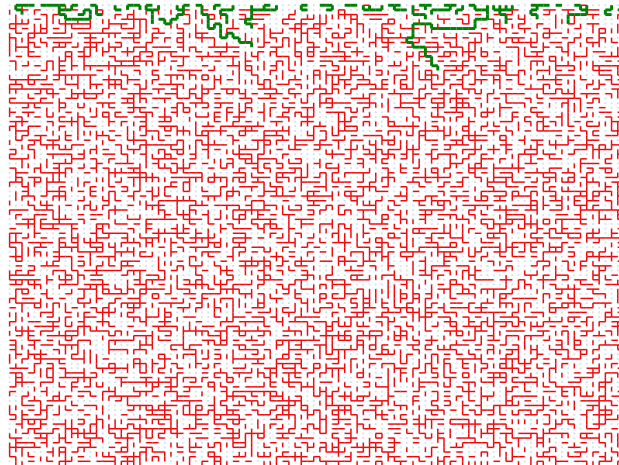
- If there is no path from  $u$  to nodes in the bottom-most layer, display the largest shortest path that originates at  $u$ .
- Otherwise, display the shortest path from  $u$  to the bottom-most layer.

---

```
l = Lattice(100)
l.percolate(0.4)
l.showPaths()
```

---

Expected output:

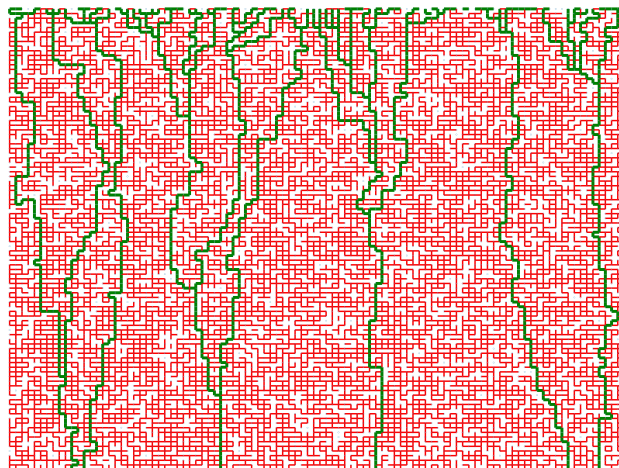


---

```
l = Lattice(100)
l.percolate(0.7)
l.showPaths()
```

---

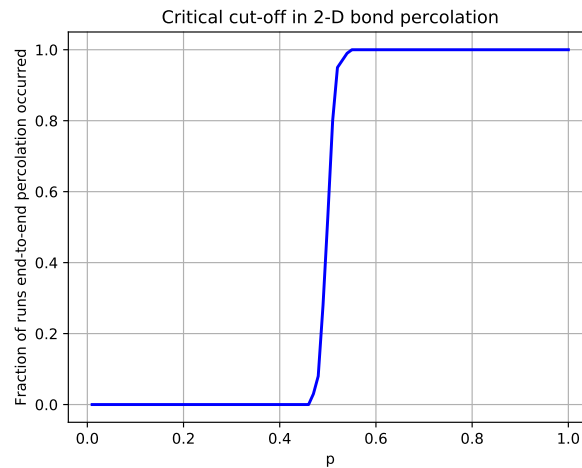
Expected output:



6. Write a function that uses the `Lattice` class to verify the following statement

[20]

*“A path exists (almost surely) from the top-most layer to the bottom-most layer of a  $100 \times 100$  grid graph only if the bond percolation probability exceeds 0.5”*



**NOTE:** The above plot is obtained by averaging outcomes of 50 runs.