Assignment 3.1 Practice Problem 1 (Build a Graph)

Problem

You are given an integer n. Determine if there is an unconnected graph with n vertices that contains at least two connected components and contains the number of edges that is equal to the number of vertices. Each vertex must follow one of these conditions:

1. Its degree is less than or equal to 1
2. It's a cut-vertex

Note

The graph must be simple.
Loops and multiple edges are not allowed.

Input format:

First line: n

Output format:

Print Yes if it is an unconnected graph. Otherwise, print No.

Constraints:

$1 <= n <= 100$

Solution:

Part 1: Visualization

In order to better understand the problem, we must first clarify and set the conditions that the graph must meet:

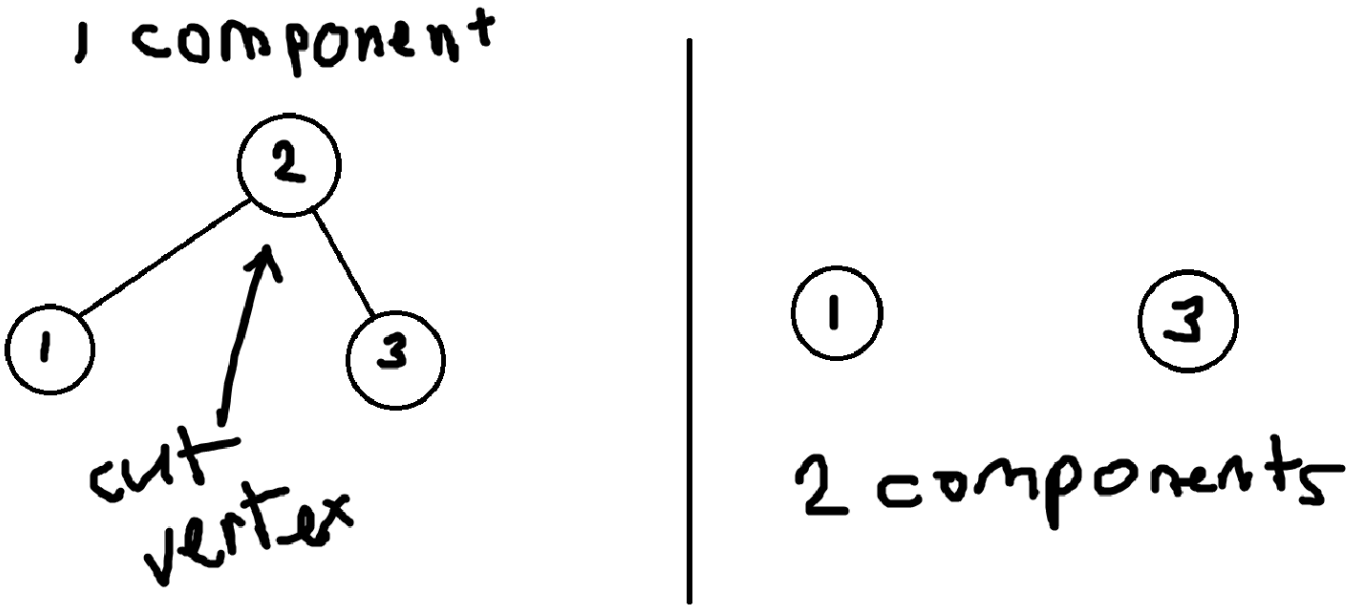
1. The number of vertices, **n**, should be equal to the number of edges.
2. The graph is unconnected (this means that for at least a pair of vertices, there should be no possible path to take in order to connect them)
3. Each vertex should either:

- have a degree of 0 or 1
- be a cut-vertex

To solve this problem, we need to determine the least number of n that would satisfy all conditions. For now, we will also ignore the 2nd condition since there is an easy way to achieve it once we get the minimum value of n.

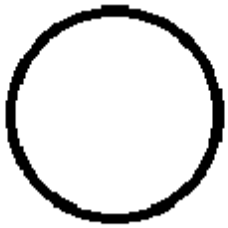
Cut-Vertex

To simplify, a cut vertex is a vertex which, when removed from the graph structure, increases the number of "components". A component is an individual graph structure that does not cover all vertices in the graph. When a cut-vertex is removed along with its incident edges, it should result in an increase in the amount of components in the graph. Refer to the figure below:



n = 1

At n = 1, we would only have 1 vertex. This means we would not have another vertex to connect to using an edge. We cannot satisfy any condition other than the 3rd, since the degree of our vertex is 0.



$v = 1$

$e = 0$

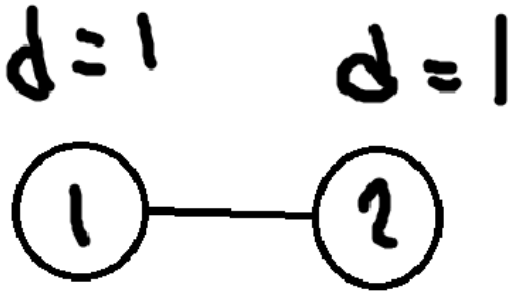
$d = 0$

n = 2

At n = 2, we can have 2 structure forms:

$v = 2$

$e = 0$



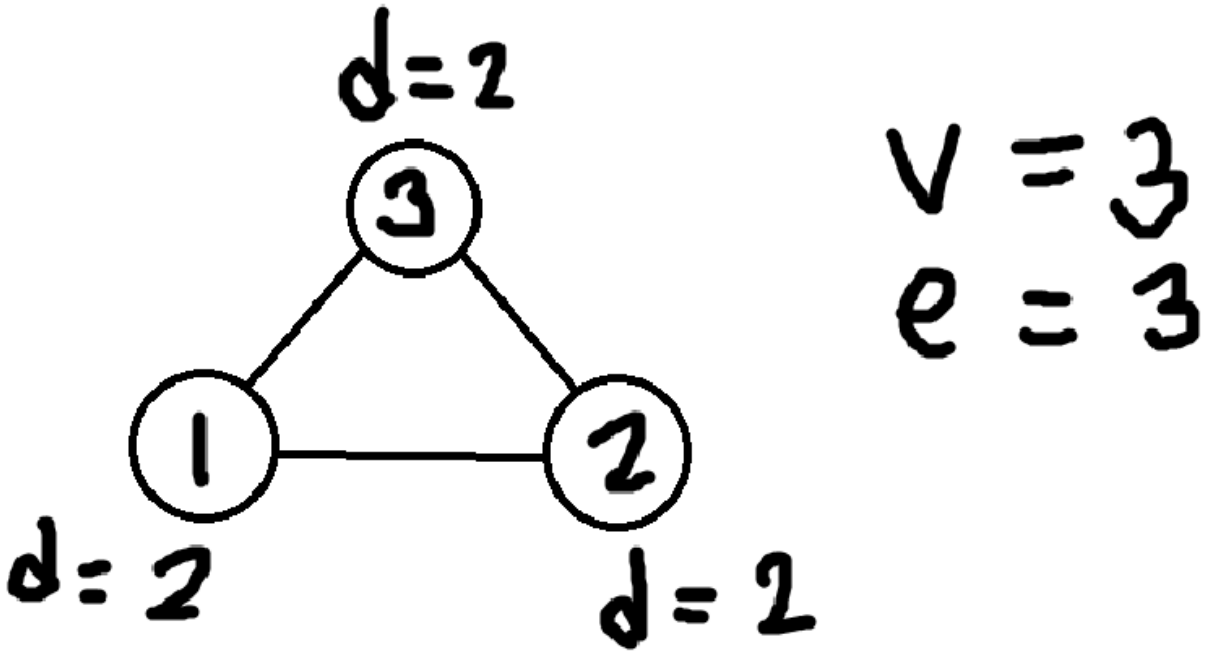
$v = 2$

$e = 0$

The maximum number of edges with 2 vertices is just 1, which does not satisfy the 1st condition set earlier. With these 2 arrangements, We can never reach the 1st condition, despite having these 2 configurations.

n = 3

By adding the 3rd vertex to the structure we have when n = 2, we can create a structure that satisfies the 1st condition:

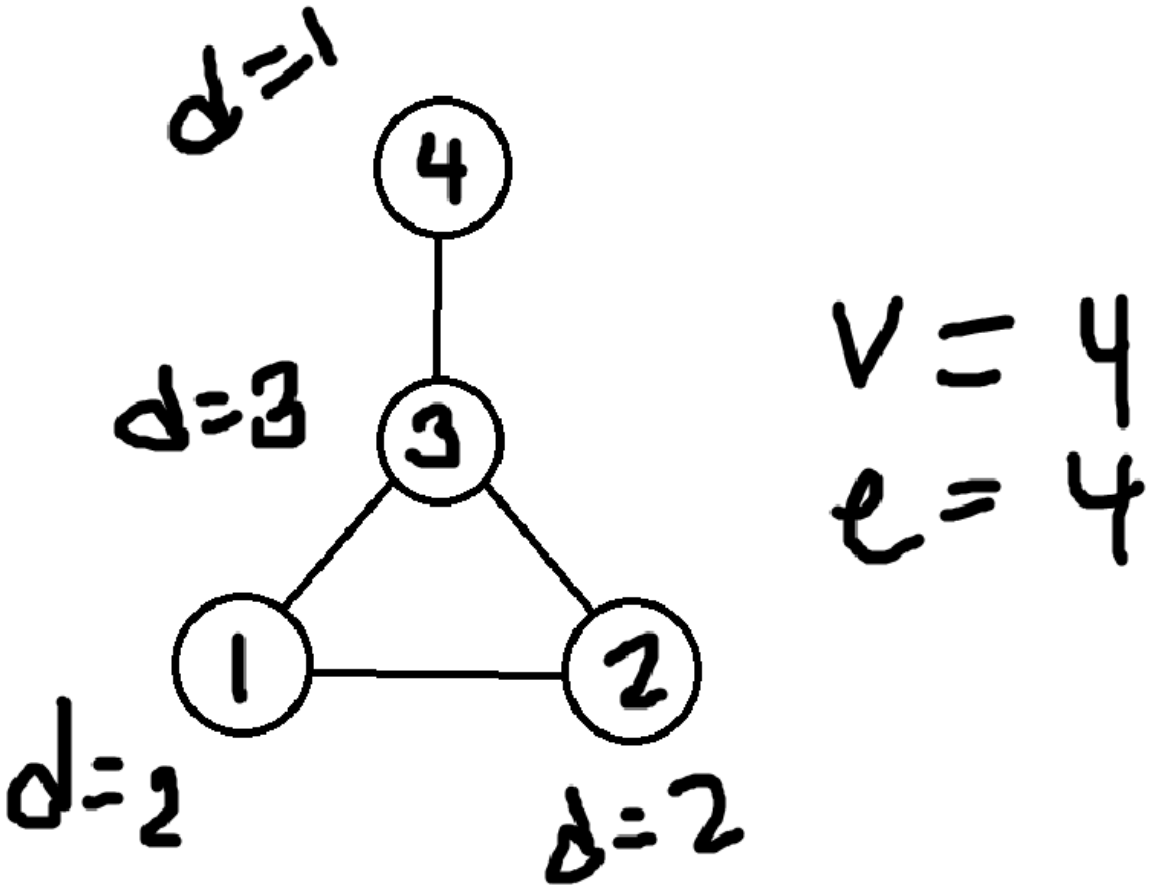


This structure allows us to have 3 vertices and 3 edges, which satisfies the 1st condition. However, all of our vertices are of degree 2, which means that they should be cut-vertices. However, removing one of these vertices do not result in an increase in the component count, since we started with 1 component and ended with 1 as well. However, achieving the 1st condition for the first time is a big step in our process of problem solving.

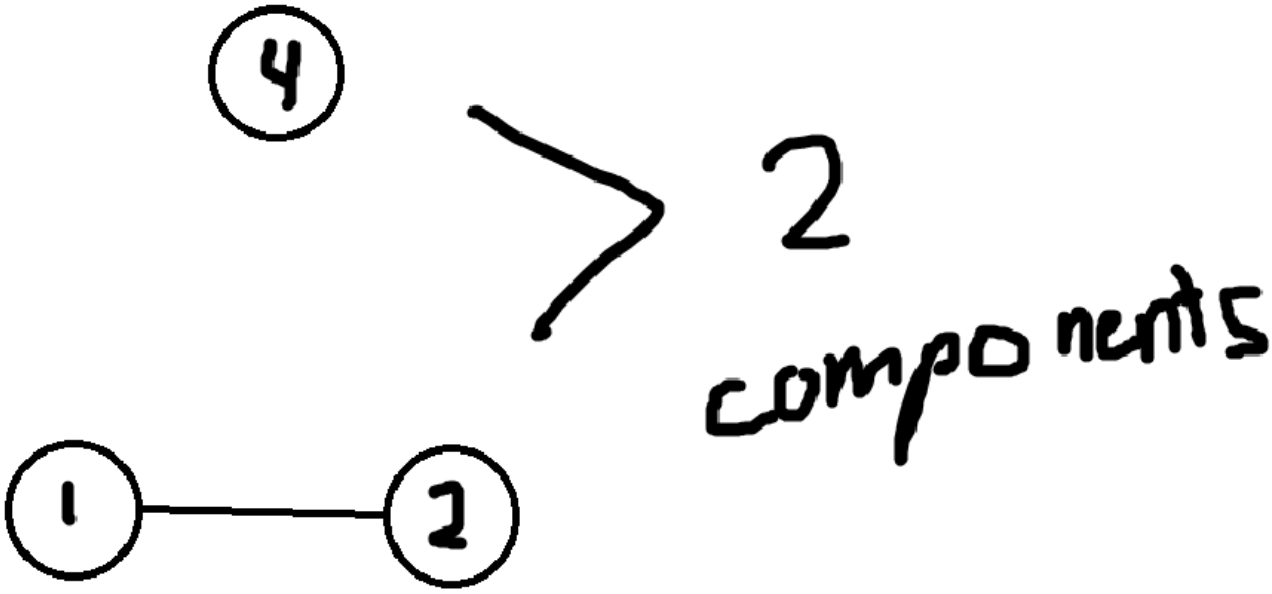
n = 4

By adding another vertex to the triangular graph structure, we maintained the equality between the vertex count and the edge count. This is because whenever we add a vertex, we only need another

edge to connect it to the main structure. It means that as long as we start with the n = 3 structure, the 1st condition will always be met.

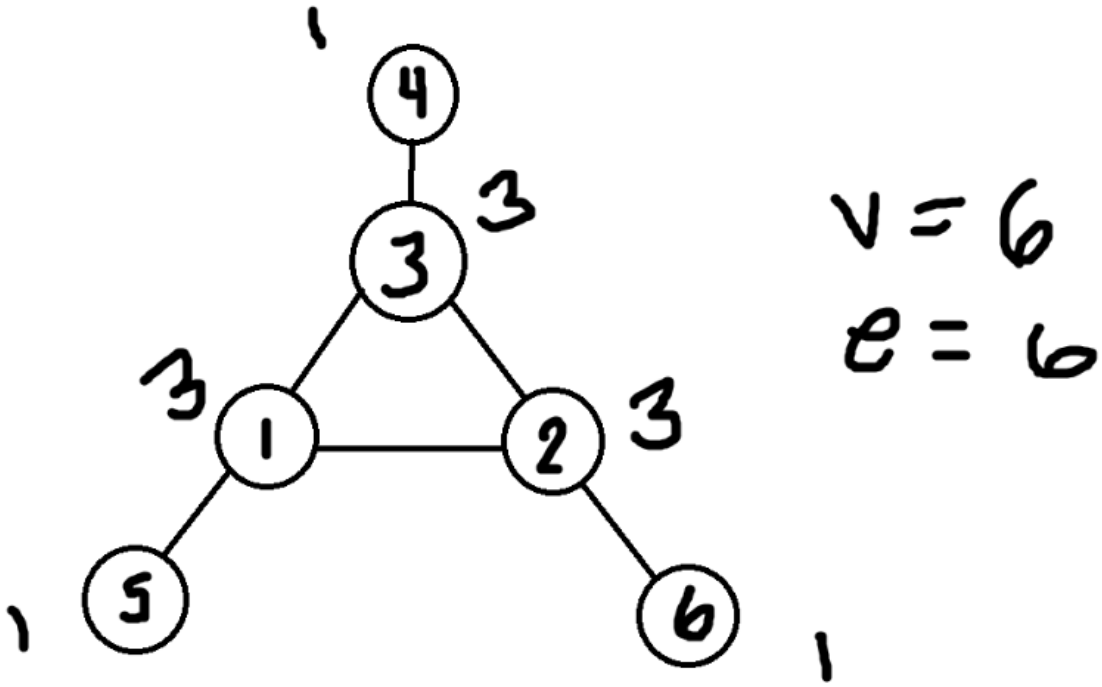


Another detail to point out is how vertex 3 is affected by the adding of the extra vertex 4. Once we connect the additional vertex, vertex 3 becomes a cut-vertex since vertex 4 and the initial triangle structure is considered as 2 separate components. This means that we can turn the vertices of the triangle form into cut-vertices by simply adding another vertex and edge to them.

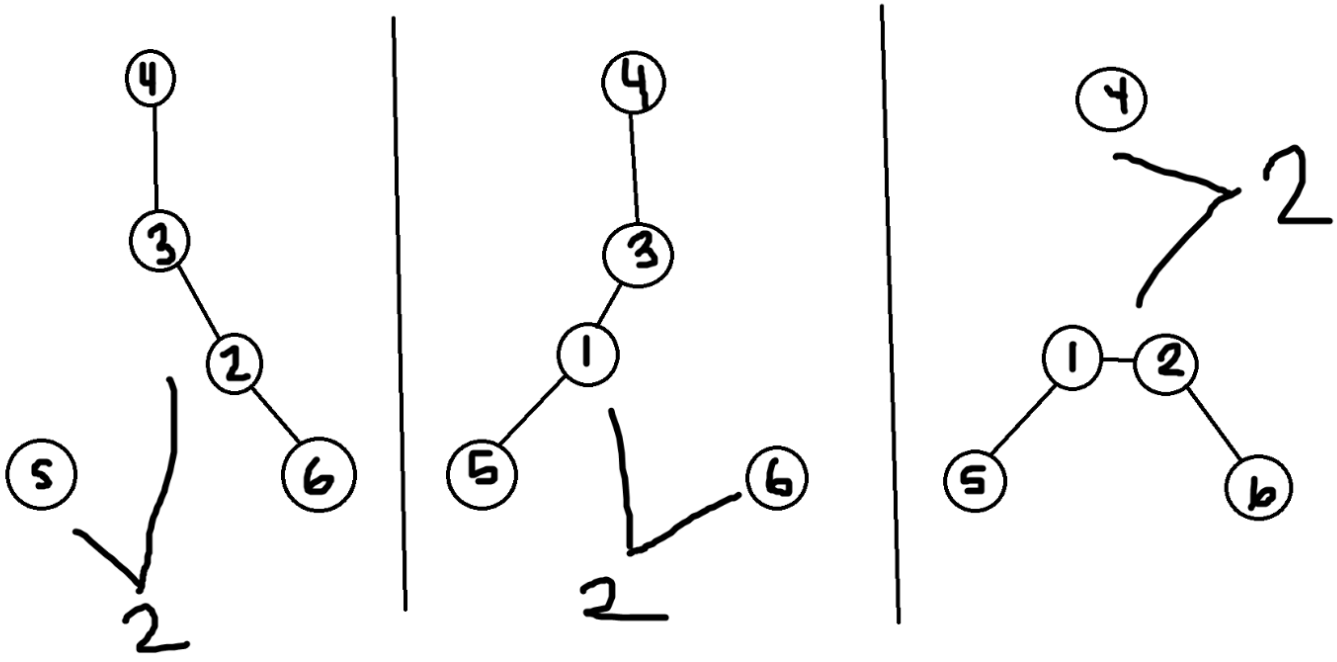


n = 6

Using the information we got from constructing the graph at n = 4, we are able to create this new structure that satisfies both the 1st condition as well as the 3rd condition completely for any vertex in the graph. By adding the additional vertex and edge similar to what we did in n = 4, we are able to meet the 2 conditions, leaving us only to deal with the 2nd condition.

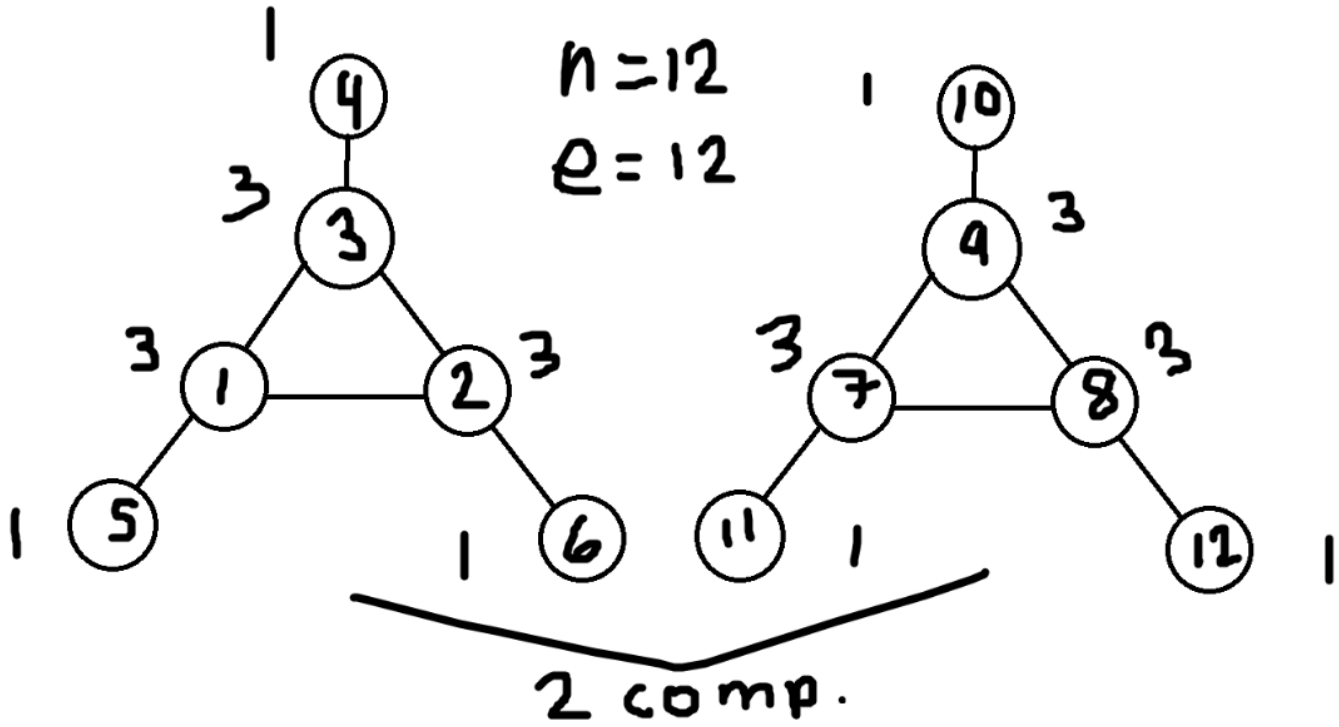


We can also double check the vertices with degrees above 1, since they should be cut-vertices. When we do so, we can see that the initial problems of the triangular structure have been solved and the first 2 conditions have been met perfectly.

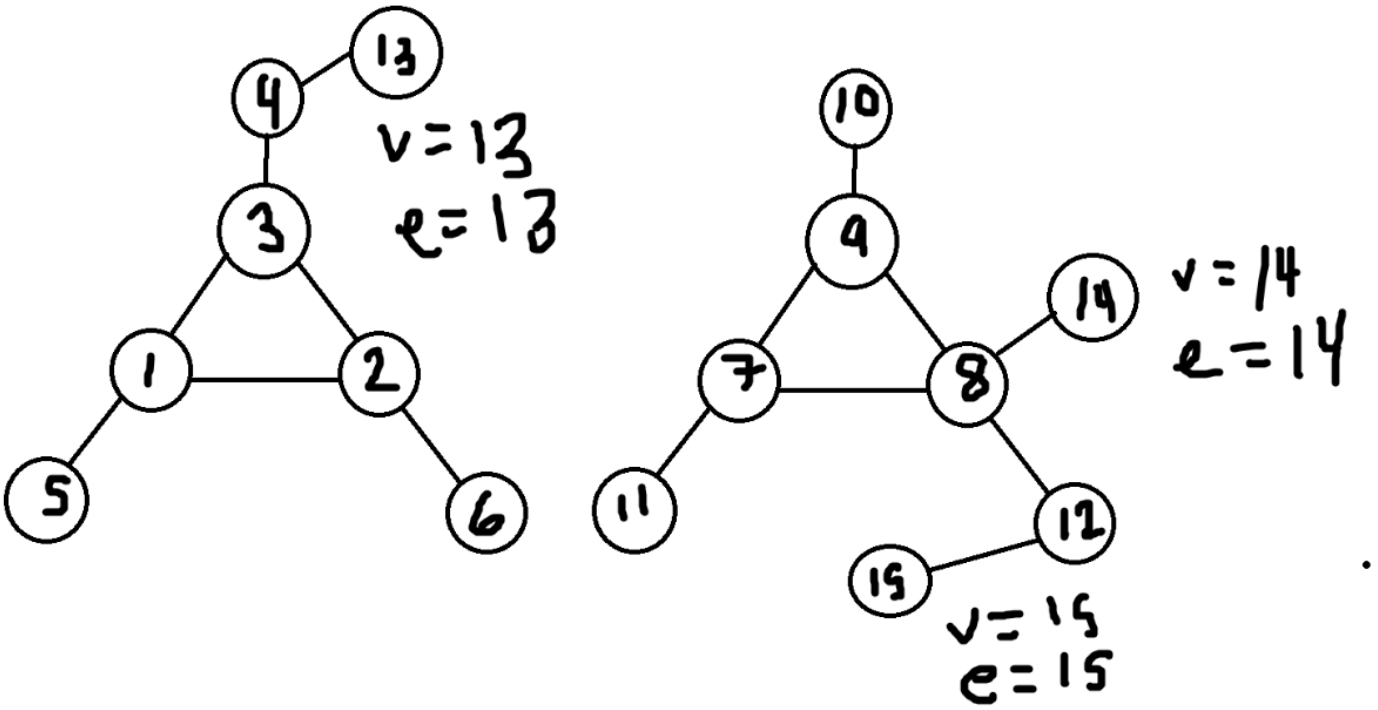


n >= 12

Now that we have the minimum value of n that satisfies both the 1st and 3rd condition, we can finally try to meet the 2nd condition, which is very simple to achieve. Since we know that the n = 6 graph can hold both condition 1 and 3, we can simply repeat the same graph as a separate component in the graph to satisfy condition 2 without breaking condition 1 and 3. Since we initially have n = 6, doubling this will give us n = 12. This is the least value of n that satisfies all 3 conditions at the same time.



Since we have this structure, we actually know that any value of n above 12 would still meet the 3 conditions. This is because of what we found out during n = 4. We found out that once the no. of edges = no. of vertices, adding one vertex requires adding 1 edge, resulting in both counts going up and remaining equal. This means that from the n = 12 graph, we can keep adding 1 vertex and 1 edge regardless of where on the graph, so we can achieve higher values of n, as well as keeping the 3 conditions satisfied.



From this, we can say that for any integer n from 1 to 100 where n ≥ 12, we can create a graph that satisfies the 3 conditions given.

Part 2: Python Implementation

For the Python Implementation of the solution above, The provided sample code for an OOP implementation of the graph will be used since it will be much easier. In order for us to be able to check if any graph will meet the condition, we will be creating some methods that will help us perform these tasks better to prove our solution as correct.

```
"""This code block will be the unmodified
    code provided in the Graph Theory
    notebook in the Discussion task."""

class Graph(object):

    def __init__(self, graph_dict=None):
        """ initializes a graph object
            If no dictionary or None is given,
            an empty dictionary will be used
        """
        if graph_dict == None:
            graph_dict = {}
        self._graph_dict = graph_dict

    def edges(self, vertice):
        """ returns a list of all the edges of a vertice"""
        return self._graph_dict[vertice]

    def all_vertices(self):
        """ returns the vertices of a graph as a set """
        return set(self._graph_dict.keys())

    def all_edges(self):
        """ returns the edges of a graph """
        return self.__generate_edges()

    def add_vertex(self, vertex):
        """ If the vertex "vertex" is not in
            self._graph_dict, a key "vertex" with an empty
            list as a value is added to the dictionary.
            Otherwise nothing has to be done.
        """
        if vertex not in self._graph_dict:
            self._graph_dict[vertex] = []

    def add_edge(self, edge):
        """ assumes that edge is of type set, tuple or list;
            between two vertices can be multiple edges!
        """
        edge = set(edge)
        vertex1, vertex2 = tuple(edge)
        for x, y in [(vertex1, vertex2), (vertex2, vertex1)]:
            if x in self._graph_dict:
                self._graph_dict[x].add(y)
            else:
                self._graph_dict[x] = [y]

    def __generate_edges(self):
        """ A static method generating the edges of the
            graph "graph". Edges are represented as sets
```

```
            with one (a loop back to the vertex) or two
            vertices
        """
        edges = []
        for vertex in self._graph_dict:
            for neighbour in self._graph_dict[vertex]:
                if {neighbour, vertex} not in edges:
                    edges.append({vertex, neighbour})
        return edges

    def __iter__(self):
        self._iter_obj = iter(self._graph_dict)
        return self._iter_obj

    def __next__(self):
        """ allows us to iterate over the vertices """
        return next(self._iter_obj)

    def __str__(self):
        res = "vertices: "
        for k in self._graph_dict:
            res += str(k) + " "
        res += "\nedges: "
        for edge in self.__generate_edges():
            res += str(edge) + " "
        return res
    def find_path(self, start_vertex, end_vertex, path=None):
        """ find a path from start_vertex to end_vertex
            in graph """
        if path == None:
            path = []
        graph = self._graph_dict
        path = path + [start_vertex]
        if start_vertex == end_vertex:
            return path
        if start_vertex not in graph:
            return None
        for vertex in graph[start_vertex]:
            if vertex not in path:
                extended_path = self.find_path(vertex,
                                               end_vertex,
                                               path)
                if extended_path:
                    return extended_path
        return None

    def count_vertices(self):
      """ returns the number of vertices
          in the graph """
      return len(self._graph_dict.keys())

```

```
    def count_edges(self):
      """ returns the number of edges of
          a given vertex in the graph """
      return len(self.__generate_edges())

    def check_cut_vertex(self, vertex):
      """ tests to see whether removing
          the given vertex and its edges
          will result into separate components,
          making it a cut-vertex """
      if len(self._graph_dict[vertex]) < 2: # checks whether the vertex has the minimum
        return 0
      edges = self.edges(vertex)
      for i in edges: # checks all vertices that are in the edge set
        check = 0
        for j in self.edges(i): # checks all the edges of the vertices obtains from fir
          if j != vertex and j in edges: # checks whether 2 edge vertices are connected
            check += 1
        if check == 0: # this means that the 2 edge vertices are not connected, thus we
          return 1 # vertex is cut-vertex
      return 0 # vertex is not cut-vertex

```