ALG2 HW 1

Quiz1

1. The following bidirectional graph G = (V,E) with V = {1,2,3,4,5,6,7,8,9} is given with an adjacency matrix:

```
  0 1 2 3 4 5 6 7 8
0 [0 1 1 0 0 0 0 0 0]
1 [1 0 1 0 0 1 0 0 0]
2 [1 1 0 0 1 0 0 0 0]
3 [0 0 0 0 1 0 0 0 0]
4 [0 0 1 1 0 0 1 0 0]
5 [0 1 0 0 0 0 1 0 0]
6 [0 0 0 0 1 1 0 1 1]
7 [0 0 0 0 0 0 1 0 0]
8 [0 0 0 0 0 0 1 0 0]
```

Given the adjacency matrix, we need to determine the minimum number of vertices that need to be colored red such that every edge has at least one red vertex.

networkx already have good approximation method for this problem, here is code:

import networkx as nx

```
adj_matrix = [
    [0, 1, 1, 0, 0, 0, 0, 0, 0],
    [1, 0, 1, 0, 0, 1, 0, 0, 0],
    [1, 1, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 1, 1, 0, 0, 1, 0, 0],
    [0, 1, 0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 1, 0, 1, 1],
    [0, 0, 0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, 0, 0]
]

G = nx.Graph()
for i in range(len(adj_matrix)):
    for j in range(len(adj_matrix[i])):
        if adj_matrix[i][j] == 1:
            G.add_edge(i + 1, j + 1)

vertex_cover = nx.approximation.min_weighted_vertex_cover(G)

print("Minimum number of vertices that need to be coloured red:", len(vertex_cover))
print("Vertices that should be coloured red:", vertex_cover)
```

a. What is the minimum number of vertices that needs to be coloured red such that the above given solution is satisfied?

b. Vertices to be colored red:

Minimum number of vertices that need to be coloured red: 6
Vertices that should be coloured red: {1, 2, 3, 5, 6, 7}

2. Does the speed of the computer can influence the complexity of an algorithm?

b. No

Algorithm complexity is universal measurement of unit operations which was indeed created to explain algorithms runtime(time, memory) without connection to certain machine

3. Given the following code

```
s = 5
while s > 0:
  s = s - 1
```

The while loop runs 5 times. In each iteration, there is one subtraction and one comparison operation. Therefore, the number of operations is 2 operations per iteration * 5 iterations = 10

4. We consider the following code that runs for a given input:

INPUT: string s[0]....s[n-1] # a string with length n

Given the code:

```
count = 0
for character in s:
  if character == 'a':
    count = count + 1
```

**a. How many “simple” operation are executed by this code on a string with length n and with a number of occurrences of ‘a’ on the input string s?

- The algorithm complexity is where is input length. More detailed

**b. What is the best possible input?

- The input with 0 elements (if such input allow it) or 1 element which is not 'a'.

**c. What is the worst possible input? And how many “simple” operations are executed in this case?

- The input = $\{s \in R^n, n = 2{,}147{,}483{,}647 \mid s[i] = 'a'\ for\ all\ i\}$
  In such case on each iteration 2 actions will be performed, so worst input is the longest possible string of 'a'. So number f operations will be

5. You are making an application that makes use of an algorithm A. When you analyse the time performances of your application which analysis of the algorithm A you need to take into account:

b. The worst case running time

6. We consider the following code that runs for a given input:

INPUT: string s[0]....s[n-1] # a string with length n

```
count = 0
for i in range(n-1):
   if s[i] == 'a':
      if s[i+1] == 'b':
         count = count + 1
```

We consider the following input sequences as given in the table below (the length of the input sequences is irrelevant) and how they influence the running time of the code. Fill in the table.

Input string	Best case	Worst case	In between
abababab...	-	Yes	-
aaaaaaaaa...	-	-	Yes
acacacacac...	Yes	-	-
bbbbbbbbb...	Yes	-	-

7. We consider the three functions f(n), g(n) and h(n) as shown in the image below, each expressing a complexity of an algorithm.

1. f(n) ∈ O(g(n))
   True
2. f(n) ∈ O(h(n))
   True
3. g(n) ∈ O(f(n))
   False
4. g(n) ∈ O(h(n))
   True
5. h(n) ∈ O(f(n))
   False

6. h(n) ∈ O(g(n))
   False

8. Fill in the table

Statement	Is it correct?	Is it the best possible bound?
$4n^2 - 300n + 12 \in O(n^2)$	Yes	Yes
$4n^2 - 300n + 12 \in O(n^3)$	Yes	No (O(n^2))
$3n + 5n^2 + 3n \in O(n^2)$	Yes	Yes
$3n + 5n^2 + 3n \in O(3n)$	No	No ()
$3n + 5n^2 + 3n \in O(4n)$	No	No ()
$50 \cdot 2^n \cdot n^2 + 5n - \log(n) \in O(2^n)$	Yes	Yes
$50 \cdot 2^n \cdot n^2 + 5n - \log(n) \in O(2.1^n)$	Yes	No ()
$50 \cdot 2^n \cdot n^2 + 5n - \log(n) \in O(2^n \cdot n^3)$	Yes	No ()

9. What is the running time of

INPUT: string s[0]....s[n-1] # a string with length n

```
count = 0
for i in range(n-1):
   if s[i] == 'a':
      if s[i+1] == 'b':
         count = count + 1
```

Running time in Big O notation:

- The running time is $O(n)$ since there is a single loop running $n-1$ times with constant time operations inside.

10. What is the running time of the following code expressed in the Big O notation?

```
result = 0
for i in range(0, n):
   for j in range(0, n):
      result = result + j
```

Running time in Big O notation:

- The running time is $O(n^2)$ since there are two nested loops each running n times.

11. Polynomial vs Exponential Running Time

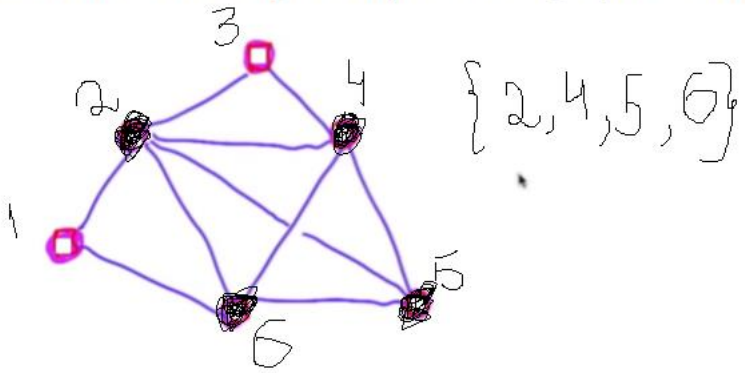
Identify if the running times are polynomial or exponential:

Running time	**Polynomial**	**Exponential**
$O(2^n * log(n))$		Yes
$O(2^{log(n)})$	Yes	
$O(1.00001^n)$		Yes
$O(n^{1000})$	Yes	
$O(2^n \cdot n^2)$		Yes

12. Largest Clique in the Graph

Clique is fully connected subgraph so it is possible to find it just by looking on it:

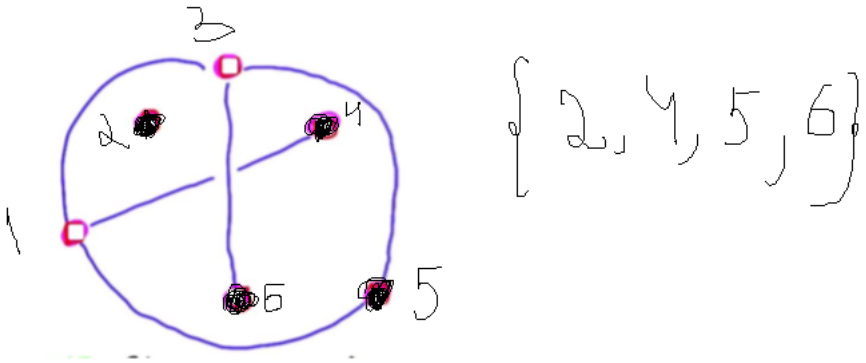
12. What is the largest clique of the graph below?



13. Largest Independent Set in the Graph

To determine the largest independent set, so looking on such small graph it is possible to find it:

13. What is the largest Independent set of the graph below?



14. What do Clique, Vertex cover and Independent Set have in common:

- **d. “Easy” to figure out if a given 0,1 mapping is valid**

Since this algorithm looks easy for human, it is hard for computer and numerically classified as NP-hard algorithm. Since it is easy to check if the given input satisfy the algorithm constarints.

Week1

week 1

generate graph

With two text boxes the user selects the number of vertices n in the graph and a probability p. Create the nxn adjacency matrix where each edge is added to the graph with probability of p.

Show the graph in a picture box.

connect

The graph may be not connected. Add a button to make it connected. In such a case: find two disconnected subgraphs, select an arbitrary vertex in each of them and add an edge between those two vertices.

Generate Graph

We need to generate a random graph based on user inputs for the number of vertices n and a probability p. This graph is represented as an adjacency matrix.

Approach

Generate Random Graph as Adj matrix:

- An adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are have conection(in variant without weights).
- For a graph with n vertices, we create matrix $A \in \mathbb{R}^{n \times n}$ initialized to zero.
- If there is an edge between vertices i and j, the matrix entry at (i,j) and (j,i) is set to 1 (since the graph is undirected).

Initialize the Graph:

- Create an empty adjacency matrix of size n×n.
- For each pair (i,j)
  - Generate a random number between 0 and 1.
  - If the random number is less than p, set the matrix entries (i,j) and (j,i) to 1 to represent conection.

```
def generate_random_graph(n, p):
    adj_matrix = np.zeros((n, n))

    for i in range(n):
        for j in range(i + 1, n):
            if random.random() < p:
                adj_matrix[i][j] = 1
                adj_matrix[j][i] = 1

    return adj_matrix
```

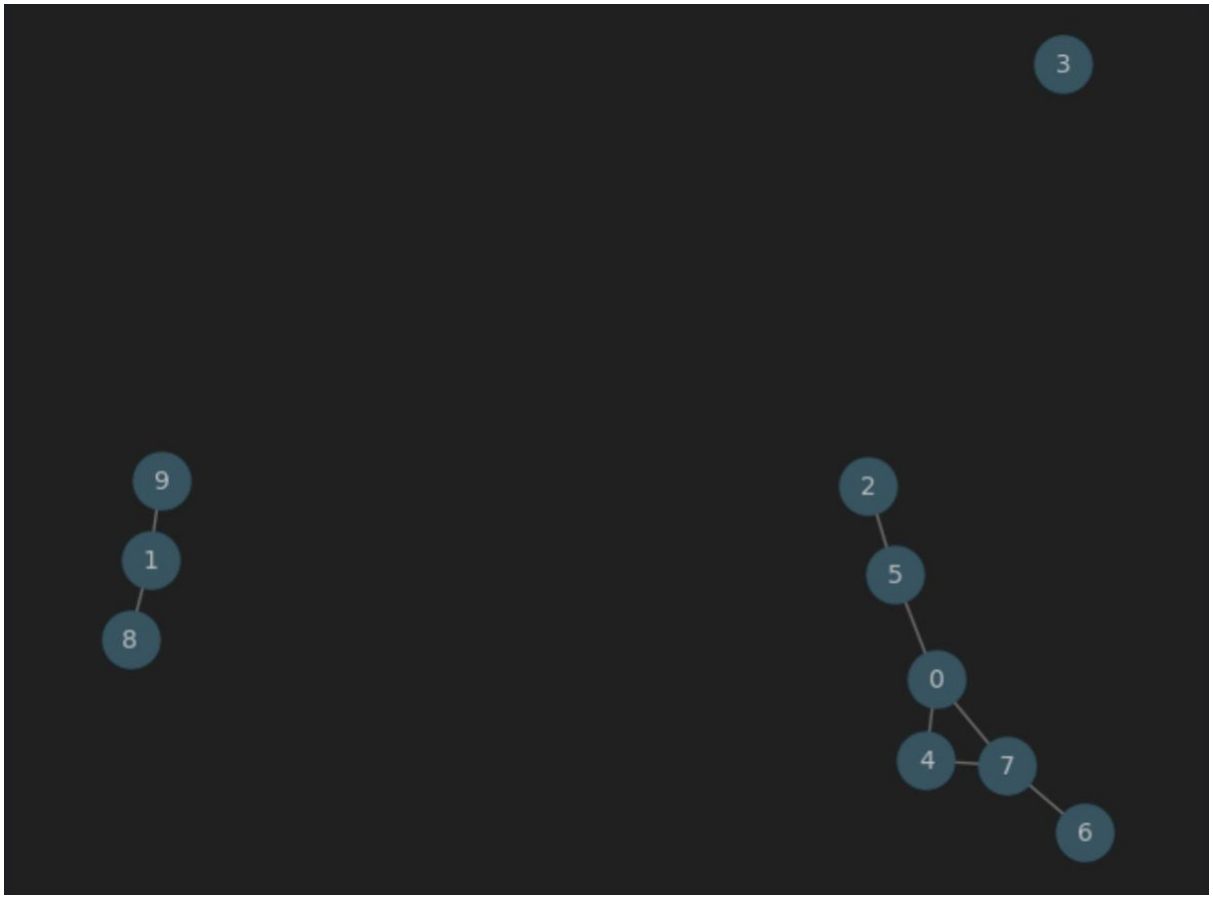
It is possible to visualize graph with networkx library:

```
def plot_graph(adj_matrix):
    G = nx.Graph(adj_matrix)
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True, node_color='skyblue', edge_color='gray', node_size=500, font_size=10)
    plt.show()
```

Test

```
n = 10
p = 0.2
adj_matrix = generate_random_graph(n, p)
plot_graph(adj_matrix)
```

And output:



Clearly the graph isnt connected which is possible due to task specification.
Now we could move to second part of assignment.

Connect

Firstly we need to define method to check if given graph(repreented as adjastency matrix is conected)

We might use BFS to travel the graph and mark each visited node as visited, then graph is connected if we visited all nodes.

BFS is implemented with queue data structure

Is Connected

```
class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if self.is_empty():
            raise IndexError("empty")
        return self.items.pop(0)
```

Basic BFS algorithm implemented with start node:

```
def bfs(graph, start_node):
    visited = set()
    queue = Queue()
    queue.enqueue(start_node)

    while not queue.is_empty():
        node = queue.dequeue()
        if node not in visited:
            visited.add(node)
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.enqueue(neighbor)

    return visited
```

Now with this helper functions we could define the method to define if graph is connected:

```
def is_connected(matrix):
    graph = nx.Graph(matrix)
    start_node = random.choice(list(graph.nodes()))
    visited = bfs(graph, start_node)

    return len(visited) == len(graph)
```

Wejust choose the random node to start and perform BFS which return visited nodes, then if visited nodes is equal to all nodes in the graph it means graph is fully connected.

Connect graph

To connect graph we need to find all not connected pieces and randomly choose nodes to set connection to 1, so then we ensure there is at least one path from disconnected subgraphs, meaning all graph is connected.

Find components

To find subgraphs, we first create a graph from the adjacency matrix. Then, we iterate through all nodes. For each unvisited node, we use it as a starting point for a BFS (Breadth-First Search). The BFS visits all connected nodes within this subgraph and adds them to the current component. The visited nodes form an array representing a disconnected subgraph.

```
def find_components(adj_matrix):
    n = len(adj_matrix)
    visited = set()
    components = []

    graph = nx.Graph(adj_matrix)

    for node in range(n):
        if node not in visited:
            component = bfs(graph, node)
            components.append(component)
            visited.update(component)

    return components
```

Conect graph

To connect a graph, we first identify all the disconnected components using the find_components function. While there is more than one component, we randomly select a node from the first component and another node from the second component, then connect these nodes by adding an edge between them in the adjacency matrix. This process merges the two components into one. We update the components list and repeat this process until there is only one component left, meaning the graph is fully connected.

```
def connect_graph(adj_matrix):
    components = find_components(adj_matrix)

    while len(components) > 1:
        comp1 = components[0]
        comp2 = components[1]
        u = random.choice(list(comp1))
        v = random.choice(list(comp2))

        adj_matrix[u][v] = 1
        adj_matrix[v][u] = 1

        G = nx.Graph(adj_matrix)
        components = list(nx.connected_components(G))

    return adj_matrix
```

So composed in solution it looks:

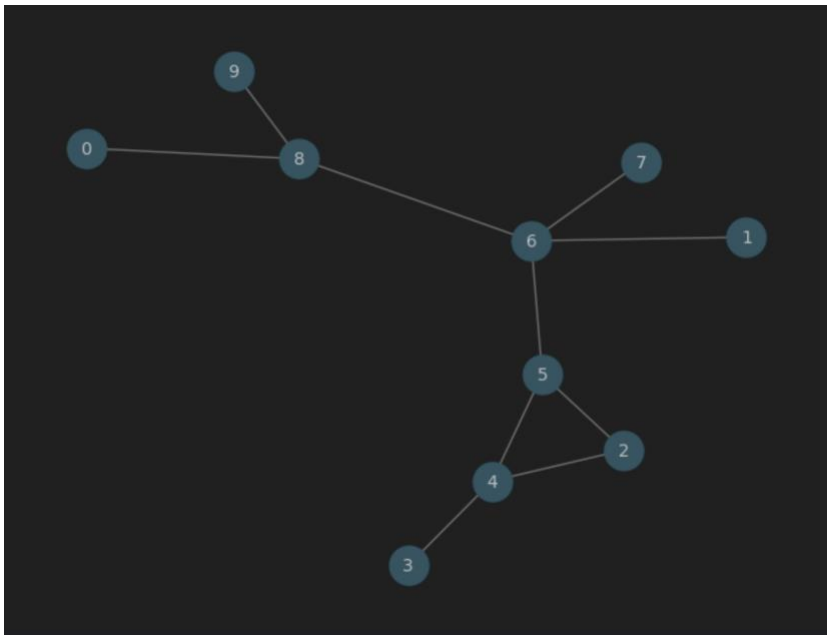
```
n = 10
p = 0.2

adj_matrix = generate_random_graph(n, p)
print("Initial randomly generated graph")
plot_graph(adj_matrix)

if not is_connected(adj_matrix):
    adj_matrix = connect_graph(adj_matrix)

print("connected graph")
plot_graph(adj_matrix)
```

Here, n is set to 10, meaning the graph will have 10 nodes. p is set to 0.2, which represents the probability of an edge being present between any two nodes in the random graph.
The generate_random_graph function generates an adjacency matrix for a random graph with 10 nodes where each possible edge is included with a probability of 0.2. The adjacency matrix adj_matrix is printed and visualized using the plot_graph function.

The is_connected function checks if the generated graph is connected. If the graph is not connected, the connect_graphfunction is called to connect all the disconnected components of the graph. This function ensures that the graph becomes fully connected by adding edges between nodes from different components.

After ensuring the graph is connected, the updated adjacency matrix is visualized again using the plot_graph function.

Result



With Adjentancy Matrix:

```
print(adj_matrix)
Executed at 2024.05.31 23:12:18 in 8ms

 [[0. 0. 0. 0. 0. 0. 0. 1. 0.]
  [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
  [0. 0. 0. 0. 1. 1. 0. 0. 0. 0.]
  [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
  [0. 0. 1. 1. 0. 1. 0. 0. 0. 0.]
  [0. 0. 1. 0. 1. 0. 1. 0. 0. 0.]
  [0. 1. 0. 0. 0. 1. 0. 1. 1. 0.]
  [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
  [1. 0. 0. 0. 0. 0. 1. 0. 0. 1.]
  [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]]
```