- Please note that one of the main goals of this course is to design efficient algorithms. So, there are points for efficiency, even though we may not explicitly state this in the question.
- Unless otherwise mentioned, assume that graphs are given in adjacency list representation.
- In the lectures, we have discussed an $O(V + E)$ algorithm for finding all the SCCs of a directed graph. We can extend this algorithm to design an algorithm `CreateMetaGraph(G)` that outputs the meta-graph of a given directed graph in $O(V + E)$ time. You may use `CreateMetaGraph(G)` as a sub-routine for this homework.
- The other instructions are the same as in Homework-1.

There are 6 questions for a total of 82 points.

---

1. A tree is defined as an undirected graph containing no cycles. An undirected graph is said to be connected iff, for every pair of vertices, there is a path between them. For this question, you have to show the following statement:

Any connected undirected graph with n vertices and $(n-1)$ edges is a tree.

We will prove the statement using Mathematical Induction. The first step in such proof is to define the propositional function. Fortunately, this is already given in the statement of the claim for this problem.

$P(n)$: Any connected undirected graph with n vertices and $(n-1)$ edges is a tree.

The base case is simple. $P(1)$ holds since any graph with 1 node and 0 edges is indeed a tree. For the inductive step, we assume that $P(1), P(2), ..., P(k)$ holds for an arbitrary $k \geq 1$, and then we will show that $P(k+1)$ holds. Consider any connected graph G with $(k+1)$ nodes and k edges. You are asked to complete the argument by doing the following:

(a) (3 points) Show that G has a node v with degree 1.

Solution: Here, G is a connected graph with $(k+1)$ nodes and k edges. *(given)*

Assume no node with degree $= 1$ exists in graph G, i.e., all $(k+1)$ nodes of G have degree < 1 or, degree > 1 in G.
Now,
$\because$ Nodes with degree $= 0$ (isolated vertices) cannot exist in G because G is a connected graph.
So, assume all nodes in G have degree > 1 which can be expressed as $(1 + \delta_i)$ (degree of ith node), where $\delta_i \geq 1, \forall i \in \{1, (k+1)\}$ and, δ_i, $i \in \mathbb{Z}$.
Now, According to Degree-Sum Formula:

$$\sum_{i=1}^{k+1} deg(v_i) = 2.|E|$$
$$\Rightarrow (1+\delta_1) + (1+\delta_2) + (1+\delta_3) + \cdots (1+\delta_{k+1}) = 2k$$
$$\Rightarrow \underbrace{(1+1) + (1+1) + (1+1) + \cdots (1+1)}_{(k+1) \text{ times}} \leq 2k \text{ , } (\because \text{ minimum value for } \delta_i \text{ will be } 1)$$
$$\Rightarrow 2.(k+1) \leq 2k$$
$$\Rightarrow 2k + 2 \leq 2k$$
$$\Rightarrow 2 \leq 0$$

But, this is not possible and hence our assumption was wrong.
$\therefore$ We can say G has at least one node v of degree 1.

Alternate Proof:
Let G' be a connected graph with k nodes and $(k-1)$ edges.
Let v be a new vertex which must be added to G' to create G.
$\because$ G must be connected and G' needs only 1 extra edge and 1 extra node to be converted to G.
$\Rightarrow$ v must be connected to any one vertex in G'.
$\therefore$ v is a node with degree 1 in G.

Q.E.D.

(b) (2 points) Consider the graph G' obtained from G by removing vertex v and its edge. Now use the induction assumption on G' to conclude that G is a tree.

Solution: Here, $G^{'}$ is a connected undirected graph with k nodes and $(k-1)$ edges and G is a connected undirected graph with $(k+1)$ nodes and k edges .
Let $P(n)$: Any connected undirected graph with n vertices and $(n-1)$ edges is a tree.
Basis Step:

For n = 1: $\because$ Any graph with 1 vertex and 0 edges is indeed a tree.
$\therefore$ For n = 1, P(n) holds.
Induction Hypothesis:

For $n = k$: Assume $P(1), P(2), \dots, P(k)$ holds true, for any arbitrary $k \geq 1$.

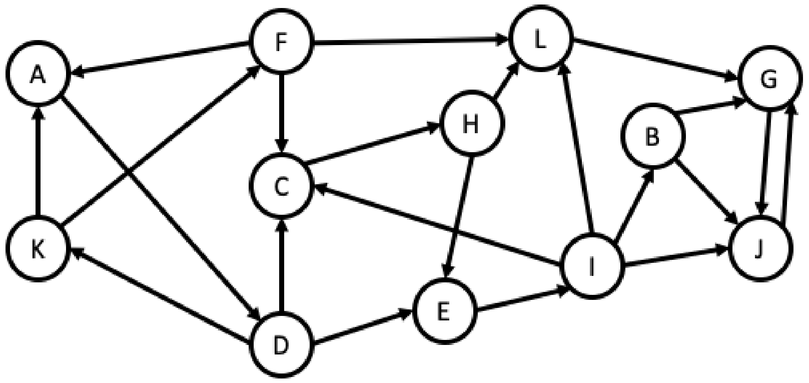
Induction Step:

For $n = k+1$: Suppose a new vertex v is added to $G^{'}$, i.e., v is an isolated vertex.
Now, $G^{'}$ must be connected.
$\Rightarrow$ We will add an edge from v (as v has degree 1) to any other vertex in tree $G^{'}$ and now we will call this obtained graph G.
$\because$ Adding an edge between tree and isolated vertex can never create a cycle and hence obtained graph is a tree.

$\because$ $[P(1) \wedge P(2) \wedge \dots \wedge P(k)] \Rightarrow P(k+1)$ holds.
$\Rightarrow$ P(n) is true, $\forall$ n $\geq$ 1 (induction step).
$\therefore$ G is a tree, i.e, a connected undirected graph with k+1 vertices and k edges.

Q.E.D.

2. Consider the following directed graph and answer the questions that follow:



(a) (½ point) Is the graph a DAG?

Solution: No, this graph is not a DAG as cycle exists, viz., $G \to J \to G$ or, $J \to G \to J$

(b) (1 point) How many SCCs does this graph have?

Solution: This graph has five strongly connected components, which are as follows: $\{G, J\}$, $\{L\}$, $\{B\}$, $\{I, C, H, E\}$ and $\{A, D, K, F\}$.

(c) (½ point) How many source SCCs does this graph have?

Solution: This graph has a single source SCC, i.e., $\{A, D, K, F\}$.

(d) (1 point) Suppose we run the DFS algorithm on the graph exploring nodes alphabetically. Given this, what is the pre-number of vertex F?

Solution: dfs (G):

Vertex	Pre	Post
A	1	24
B	7	12
C	3	18
D	2	23
E	5	14
F	20	21
G	8	11
H	4	17
I	6	13
J	9	10
K	19	22
L	15	16

$\Rightarrow \text{Pre}(F) = 20$

(e) (1 point) Suppose we run the DFS algorithm on the graph exploring nodes alphabetically. Given this, what is the post-number of vertex J?

Solution: dfs (G):

Vertex	Pre	Post
A	1	24
B	7	12
C	3	18
D	2	23
E	5	14
F	20	21
G	8	11
H	4	17
I	6	13
J	9	10
K	19	22
L	15	16

$\Rightarrow \text{Post}(J) = 10$

(f) (1 point) Is it possible to add a single edge to this graph so that the graph becomes a strongly connected graph? If so, which edge would you add?

Solution: Yes, it is possible to add a single edge to this graph so that the graph becomes a strongly connected graph.
Adding one edge from any vertex in sink SCC ({G,J}) to any vertex in source SCC ({A,D,K,F}) of meta graph of G, will make Graph G, a strongly connected graph.

3. You are given a directed graph $G = (V, E)$ in which each node $u \in V$ has an associated price, denoted by $price(u)$, which is a positive integer. The cost of a node u, denoted by $cost(u)$, is defined to be the price of the most expensive node reachable from u (including u itself). Your goal is to compute the cost of every node in the graph (this can be produced as an array *cost* of size $|V|$).

   (a) (9 points) Give a linear time algorithm that works for DAG's. (*Hint: Handle the vertices in a particular order*)

Solution:

Input: DAG $G = (V, E)$; node $v \in V$, count of v starts at 1 and, an array prices of size $|V|$.

Output: Cost Array.

Algorithm: Procedure solve(G, prices[ ])

```
1. Find the Topological Sort of given graph G.

2. For nodes in topological order:
       - cost[lastnode] = price[lastnode].
       - for i = second last node to first node:
             -Initialize a variable maxcost_of_neighbors = 0.
             - for all neighbors n of i:
                   maxcost_of_neighbors = max(maxcost_of_neighbors, price[n])
             - cost[i] = max(cost[i], maxcost_of_neighbors)

3. Return Cost Array.
```

Note: In the above algorithm:

→ *= means assignment.*

→ *Indentation denotes the scope.*

→ *max is a function that returns the maximum of two values in $O(1)$ time, using if-else statements.*

Running Time Analysis:

1. Finding the Topological Sort of given graph G takes $O(V + E)$ time in total, using DFS.
2. For nodes in topological order, we are first initializing the cost of its last node equal to its price which takes $O(1)$ time. Then for the remaining nodes, we find the max cost of all its neighbors by traversing its adjacency list which takes $O(deg_{node})$ time, followed by some constant operations. In total, we are just traversing the adjacency list of G, which takes a total of $O(V + E)$ time.
3. Finally we return the output in $O(1)$ time.

$$\begin{aligned}
\text{T.C.} &= O\left(|V| + |E| + |V| + |E| + 1\right) \\
&= O\left(2.|V| + 2.|E| + 1\right) \\
&= O\left(c + |V| + |E|\right), \text{ where } c \text{ is some constant}
\end{aligned}$$

Proof of Correctness:

Termination Check: Finding the topological sort of graph uses DFS which calls Explore(G, u) on every vertex u, exactly once and terminates. This is because once a node is marked visited, it will never be explored again. Then, the next step includes a simple traversal of the adjacency list of Graph, which takes finite time before termination. Hence, the algorithm eventually terminates.

Proof: We know that every DAG has at least one source and at least one sink. Now, the sink will never have any neighbors. So, the cost of every sink node will be equal to its price, which is done by our algorithm for at least one sink node. Now, for every other node, its cost will be equal to the maximum of the cost of its neighbors and its cost. And this will be calculated correctly as we are calculating costs in reverse order of topological order. This is because:
Suppose a to b is an edge, i.e., b is a neighbor of a. Then a will definitely come before b in topological sort. So, the cost of b will be computed before the cost of a, and traversing from last to first will ensure that the cost of a is calculated correctly.
$\therefore$ Our Algorithm is correct.

Q.E.D.

(b) (9 points) Extend this to a linear time algorithm that works for any directed graph. (*Hint: Consider making use of the meta-graph of the given graph.*)

Give running time analysis and proof of correctness for both parts.

Solution:

Input: Graph $G = (V, E)$; node $v \in V$, count of v starts at 1 and, an array prices of size $|V|$.

Output: Cost Array.

Algorithm: Procedure solve(G, prices[ ])

```
1. Initialize three arrays max_SCC, cost_SCC and, cost of size |V| with all 0's.

2. G' = CreateMetaGraph(G) with augmentation that we will take a variable large = 0 and
   whenever DFS reaches a node we will keep updating large with max(large, price of current
   node) and whenever we find an SCC, i.e., vertex of G', we will set max_SCC[current node]
   = large.

3. Find Topological Sort of MetaGraph G'.

4. For each node (SCC) in topological order:
       - for i = last node to first node:
           -Initialize a variable maxcost_of_neighbors = 0.
           - for all neighbors n of i:
               maxcost_of_neighbors = max(maxcost_of_neighbors, max_SCC[n])
           - cost_SCC[i] = max(cost_SCC[i], maxcost_of_neighbors)

5. Run DFS on vertices (SCC) of G' and for all vertices (inside SCC) i, set cost[i] =
   cost_SCC[i].

6. Return Cost Array.
```

Note: *In the above algorithm:*

→ *= means assignment.*

→ *Indentation denotes the scope.*

→ *max is a function that returns the maximum of two values in $O(1)$ time, using if-else statements.*

Running Time Analysis:

1. Initializing an array of size $|V|$ will take $O(V)$ time.
2. Creating a MetaGraph with augmentation takes $O(V + E)$ time.
3. Assume that the meta graph has V' nodes and E' edges. Finding the Topological Sort of each connected component of given graph G takes $O(V' + E')$ time in total, using DFS.
4. For each node (SCC) in topological order, we find the max cost of all its neighbors by traversing its adjacency list which takes $O(degv)$ time, followed by some constant operations. In total, we are just traversing the adjacency list of G', which takes a total of $O(V' + E')$ time.
5. Then we will again apply DFS on Metagraph G' and set the cost of a node equal to the cost of the SCC it belongs to which takes $O(V + E)$ time.
6. Finally we return the output in $O(1)$ time.

$$\begin{aligned}
\text{T.C.} &= O\left(|V| + |V| + |E| + |V|' + |E|' + |V|' + |E|' + |V| + |E| + 1\right) \\
&= O\left(3.|V| + 2.|E| + 2.|V|' + 2.|E|' + 1\right) \\
&= O\left(c + |V| + |E|\right), \text{ where } c \text{ is some constant}
\end{aligned}$$

Proof of Correctness:

Termination Check: Creating a meta graph uses DFS (even with augmentation) and Finding the topological sort of graph uses DFS which calls Explore(G, u) on every vertex u, exactly once and terminates. This is because once a node is marked visited, it will never be explored again. Then, the next step includes a simple traversal of the adjacency list of Graph, which takes finite time before termination. Hence, the algorithm eventually terminates.

Proof: The Metagraph G' will surely be a DAG. Again, we know that every DAG has at least one source and at least one sink. Now, for every other node (SCC), its cost will be equal to the maximum of the cost of its neighbors and its cost. And this will be calculated correctly as we are calculating costs in order of topological order. This is because:
Suppose a to b is an edge, i.e., b is a neighbor of a. Then a will definitely come before b in topological sort. So, the cost of b will be computed before the cost of a, and traversing from last to first will ensure that the cost of a is calculated correctly. Also, since every node in a SCC is strongly connected. Hence, the cost of each node will be equal to the cost of SCC it belongs to.
∴ Our Algorithm is correct.

Q.E.D.

4. Given a directed graph $G = (V, E)$ that is not a strongly connected graph, you have to determine if there exists a pair of vertices $u, v \in V$ such that the graph $G' = (V, E \cup \{(u, v)\})$ is strongly connected. In other words, you must determine if there exists a pair of vertices $u, v \in V$ such that adding a directed edge from u to v in G converts it into a strongly connected graph. Design an algorithm for this problem. Your algorithm should output "yes" if such an edge exists and "no" otherwise.

   (a) (9 points) Give a linear time algorithm for DAGs.

Solution:

Input: Graph $G = (V, E)$; node $v \in V$, count of v starts at 1.

Output: Either "yes" or "no".

Algorithm: Procedure solve(G)

```
1. Initialize two arrays - in_deg and out_deg, of size |V| with all 0's.
2. For all v in V:
        - Initialize count = 0.
        - For all 'node' in adj[v]:
                - Increment in_deg[node].
                - Increment count.
        - Set out_deg[v] = count.
3. Initialize two variables - source = 0 and sink = 0.
4. For all v in in_deg and out_deg:
        - if (in_deg[v] == 0) increment source.
        - if (out_deg[v] == 0) increment sink.
5. If (source == 1 and sink == 1)
        output "yes"
   else
        output "no"
```

Note: In the above algorithm:

→ *= means assignment.*

→ *== means equality comparison operator.*

→ *Indentation denotes the scope.*

→ *max is a function that returns the maximum of two values in $O(1)$ time, using if-else statements.*

Running Time Analysis:

1. Initializing both arrays with 0 takes $O(V)$ time.
2. For each vertex - we will initialize a count variable equal to 0 that takes $O(1)$ time. Then, we will traverse the vertex's adjacency list and perform some O(1) operations, which takes $deg(v)$ time, for any vertex v. Finally, we will set out-degree, which takes $O(1)$ time. In totality, we are just traversing the entire adjacency list once, which takes $O(V + E)$ time.
3. Then we will initialize two variables source and sink to 0 which takes $O(1)$ time.
4. Then we will traverse in_deg and out_deg arrays and perform some constant operations, which will take $O(V)$ time.

5. Finally, we will perform a simple if-else comparison before returning the output, which will take $O(1)$ time.

$$\begin{aligned}\therefore \text{T.C.} &= O\left(|V| + |V| + |E| + |V| + 1\right)\\ &= O\left(1 + 3.|V| + |E|\right)\\ &= O\left(c + |V| + |E|\right), \text{ where } c \text{ is some constant}\end{aligned}$$

Proof of Correctness:

Termination Check: Finding the indegrees and outdegrees involves a simple traversal of the finite adjacency list which terminates. Then, the algo simply traverses the indegree and outdegree arrays, which runs V iterations and then returns the answer. Hence, the algorithm eventually terminates.

Proof: We know that every DAG has at least one source and at least one sink. Also, the indegrees of the source must be 0, and the outdegrees of a sink must be 0. If exactly one source and one sink exist in any graph, then we can connect the sink with the source, using one directed edge, thereby converting the graph into a single SCC as every node in the graph is reachable from some source.
$\therefore$ Our Algorithm is correct.

Q.E.D.

(b) (9 points) Extend this to a linear time algorithm that works for any directed graph. (*Hint: Consider making use of the meta-graph of the given graph.*)

Give running time analysis and proof of correctness for both parts.

Solution:

Input: Graph $G = (V, E)$; node $v \in V$, count of v starts at 1.

Output: Either "yes" or "no".

Algorithm: Procedure solve(G)

```
1. G' = CreateMetaGraph(G).
2. Initialize two arrays - in_deg and out_deg, of size $|V|$ with all -1's.
3. For all v in the adjacency list of G':
        - Initialize count = 0.
        - For all 'node' in adj[v]:
            - Increment in_deg[node].
            - Increment count.
        - Set out_deg[v] equal to count.
4. Initialize two variables - source = 0 and sink = 0.
5. For all v in in_deg and out_deg:
        - if (in_deg[v] == 0) increment source.
        - if (out_deg[v] == 0) increment sink.
```

```
6. If (source == 1 and sink == 1)
           output "yes"
   else
           output "no"
```

Note: In the above algorithm:

→ *= means assignment.*

→ *== means equality comparison operator.*

→ *Indentation denotes the scope.*

Running Time Analysis:

1. Creating a MetaGraph takes $O(V + E)$ time. Suppose it returns a meta graph with V' vertices and E' edges.
2. Initializing both arrays with -1 takes $O(V)$ time.
3. For each vertex *(SCC)* in G'- we will initialize a count variable equal to 0 that takes $O(1)$ time. Then, we will traverse the vertex's adjacency list and perform some O(1) operations, which takes $deg(v)$ time, for any vertex (SCC) v. Finally, we will set out-degree, which takes $O(1)$ time. In totality, we are just traversing the entire adjacency list of metagraph G' once, which takes $O(V' + E')$ time.
4. Then we will initialize two variables source and sink to 0 which takes $O(1)$ time.
5. Then we will traverse in_deg and out_deg arrays and perform some constant operations, which will take $O(V)$ time.
6. Finally, we will perform a simple if-else comparison before returning the output, which will take $O(1)$ time.

$$\begin{aligned}\therefore \text{T.C.} &= O\left(|V| + |E| + |V| + |V|' + |E|' + |V| + 1 + |V| + 1\right)\\ &= O\left(2 + 4.|V| + |E| + |V|' + |E'\right)\\ &= O\left(c + |V| + |E|\right), \text{ where } c \text{ is some constant}\end{aligned}$$

Proof of Correctness:

Termination Check: Creating Metagraph takes a DFS traversal which terminates once all nodes are visited. Finding the indegrees and outdegrees involves a simple traversal of the finite adjacency list which terminates. Then, the algo simply traverses the indegree and outdegree arrays, which runs V iterations and then returns the answer. Hence, the algorithm eventually terminates.

Proof: We know that a metagraph of a graph is a DAG. Also, We know that every DAG has at least one source and at least one sink. Also, the indegrees of the source must be 0, and the outdegrees of a sink must be 0. If exactly one source and one sink exist in any graph, then we can connect the sink with the source, using one directed edge, thereby converting the graph into a single SCC as every node in the graph is reachable from some source and each vertex in G' is already an SCC already.
∴ Our Algorithm is correct.

Q.E.D.

5. (18 points) A directed graph $G = (V, E)$ is called one-way-connected if, for all pair of vertices u and v, there is a path from vertex u to v **or** there is a path from vertex v to u. Note that the "or" in the previous statement is a logical OR, not XOR. Design an algorithm to check if a given graph is one-way-connected. Give running time analysis and proof of correctness.

Solution:

Input: Graph $G = (V, E)$; node $v \in V$, count of v starts at 1.

Output: Either "one-way connected" or "not one-way connected".

Algorithm: Procedure solve(G)

```
1. G' = CreateMetaGraph(G).
2. Find Topological Sort of MetaGraph G'(V',E').
3. count = 0.
4. For each pair of adjacent node (u,v) in topological sort starting from the first two nodes (u,v)
   to last two nodes:
           - Check whether an edge exists from u to v or v to u.
           - If an edge does not exist:
                   - count = 1
                   - break the loop
5. If count is equal to 1:
           - Graph is not-one way connected.
   Else :
           - Graph is one-way conneceted.
```

Note: In the above algorithm:

→ *Indentation denotes the scope.*

Running Time Analysis:

1. Creating a MetaGraph takes $O(V+E)$ time. Suppose it returns a meta graph with V' vertices and E' edges.
2. Finding the Topological Sort of given graph G takes $O(V'+E')$ time in total, using DFS which is less than $O(V+E)$.
3. For each pair of adjacent node (u,v) in topological sort we see whether there is a path from u to v or v to u for which we perform O(1) time operation. So total time taken will be $O(V'-1)$.

$$\begin{aligned}
\therefore \text{T.C.} &= O\left(|V| + |E| + |V'| + |E'| + |V'| - 1\right) \\
&= O\left(|V| + |E| + 2|V'| + 2|E'| - 1\right) \\
&= O\left(|V| + |E|\right)
\end{aligned}$$

Proof of Correctness:

Termination Check: Finding the topological sort of graph uses DFS which calls Explore(G, u) on every vertex u, exactly once and terminates. This is because once a node is marked visited, it will never be explored again. Then, the next step includes a simple traversal of the topological Sort of MetaGraph pairwise, which takes finite time before termination. Hence, the algorithm eventually terminates.

Proof: First we will find the Metagraph G' with V' vertices of the given graph G with V vertices and then find the topological sort of the Resulting graph.

P(V'): A directed acyclic graph G' with V' vertices will be one way connected if there exists a edge between every adjacent node in the Topological sort of G'.

Base Step : For V' = 1 ,a graph with one node will always be one way connected.

Induction hypothesis : Let us assume for V' = k , A directed acyclic graph with first k nodes in the Topological Order will be one way connected for $k < V'$ and $k \geq 1$.

Inductive Step : In the Topological Sort we will take the next (k+1) th vertex and as we have assumed Graph till kth vertex in Topological sort is one-way connected and if there exists a edge between (k+1) th vertex and Graph till kth vertex then:
Graph with k vertices union (k+1) th vertex in Topological sort will be one way connected.
$\therefore P(k) \rightarrow P(k+1)$
Hence from the inductive step we can say that Graph with V' vertices is one way connected .

The components of a Metagraph are strongly connected and hence we can say the vertices in a strongly connected component are also one-way connected. We know Metagraph are DAG and above we have prooved that DAG are one way connected if in the Topological Sort of G there is a edge between every adjacent node.
Hence proved our algorithm is correct.

Q.E.D.

6. (18 points) Design an algorithm that given a directed acyclic graph $G = (V, E)$ and a vertex $u \in V$, outputs all the nodes that have a simple path to u with a length that is a multiple of 3. *(Recall that a simple path is a path with all distinct vertices.)*

   Give running time analysis and proof of correctness.

Solution:
Input: DAG $G = (V, E)$; node $v \in V$, count of v starts at 1 and a target node u.

Output: All the nodes that have a simple path to u with a length that is a multiple of 3.

Algorithm: Procedure solve(G, u)

1. Create G^R.
2. Initialize a count = 0 and array ans.
3. Start BFS, but without maintaining any visited array, from u and keep incrementing count, whenever (count%3 == 0), add that node to array ans.
4. Sort the array ans and remove duplicates.
5. Return the array ans.

Note: *In the above algorithm:*

→ *= means assignment.*

→ *== means equality comparison operator.*

→ *Indentation denotes the scope.*

Running Time Analysis:

1. Creating G^R takes $O(V + E)$ time.
2. Calling BFS in a DAG will take almost $O(V + E)$ time.
3. Sorting the array will take $O(V.logV)$ time.
4. Returning the array will take $O(1)$ time.

$$\begin{aligned}\therefore \text{T.C.} &= O\left(|V| + |E| + |V| + |E| + |V logV| + 1\right)\\ &= O\left(2.|V| + 2.|E| + |V logV| + 1\right)\\ &= O\left(|V| log|V| + |E|\right)\end{aligned}$$

Proof of Correctness: Since, BFS travels a graph level-wise, i.e., it first explores neighbors at distance 1, then nodes at distance 2, then nodes at distance 3, and so on. And since, we are not using the visited array, a node that is reachable from paths of different lengths, will be considered by our BFS algo, for all possible paths, thereby outputting all nodes at a distance at a multiple of 3 in the graph from the target node u.
$\therefore$ Our Algorithm is correct.

Q.E.D.