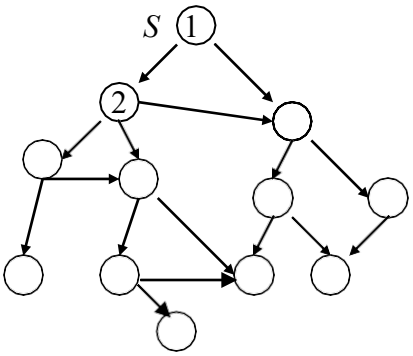
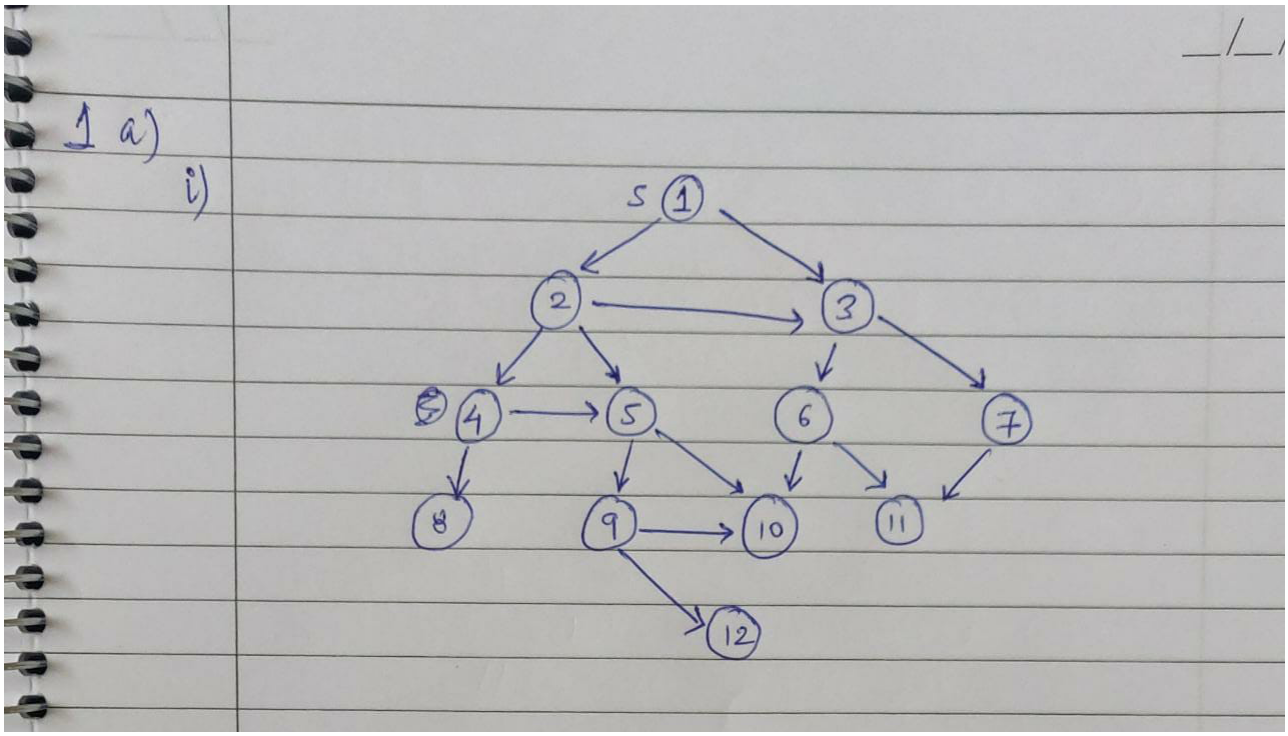
Design and Analysis of Algorithms
CS 575, Spring 2023

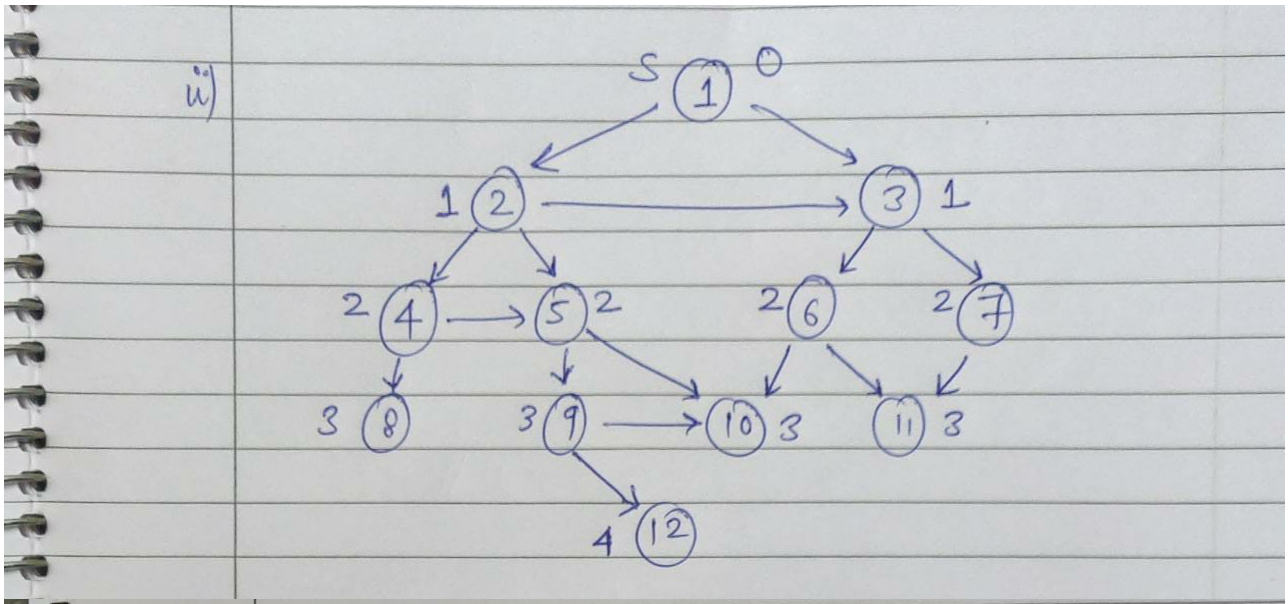
Theory Assignment 3.2

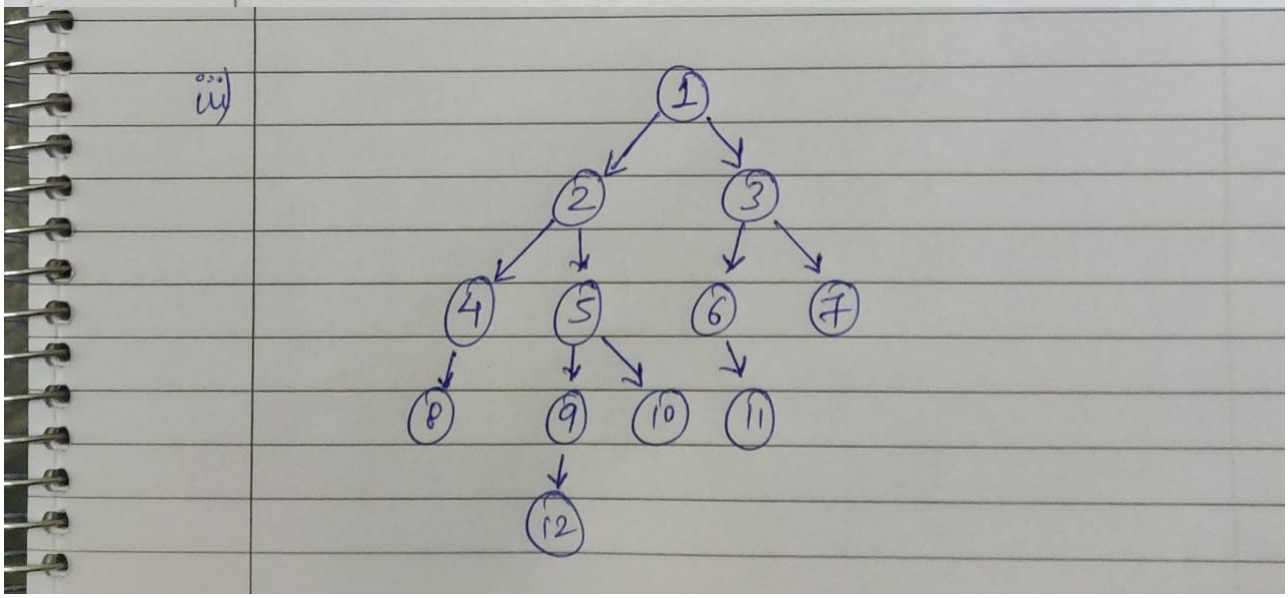
Due on 4/13/23 (Thursday) at 11:59pm

1. [20 points] In this problem, you will apply different traversal methods on a given directed graph. The start node is denoted by S. When arbitrary decisions on order must be made assume that child nodes are visited from left to right.
   a. [10 points] Perform a breadth-first search on the directed graph below (on the left). (i) *Number the nodes* according to the order in which they are visited (become gray). For example, the first node visited is *S* so *S* is numbered 1. Then, the left child of *S* is visited so it is numbered 2. Show the order numbers inside the circles. (ii) Show the distance of each node to *S* beside each circle. (iii) Show the breadth-first tree.

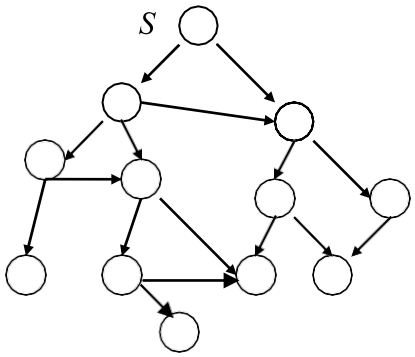


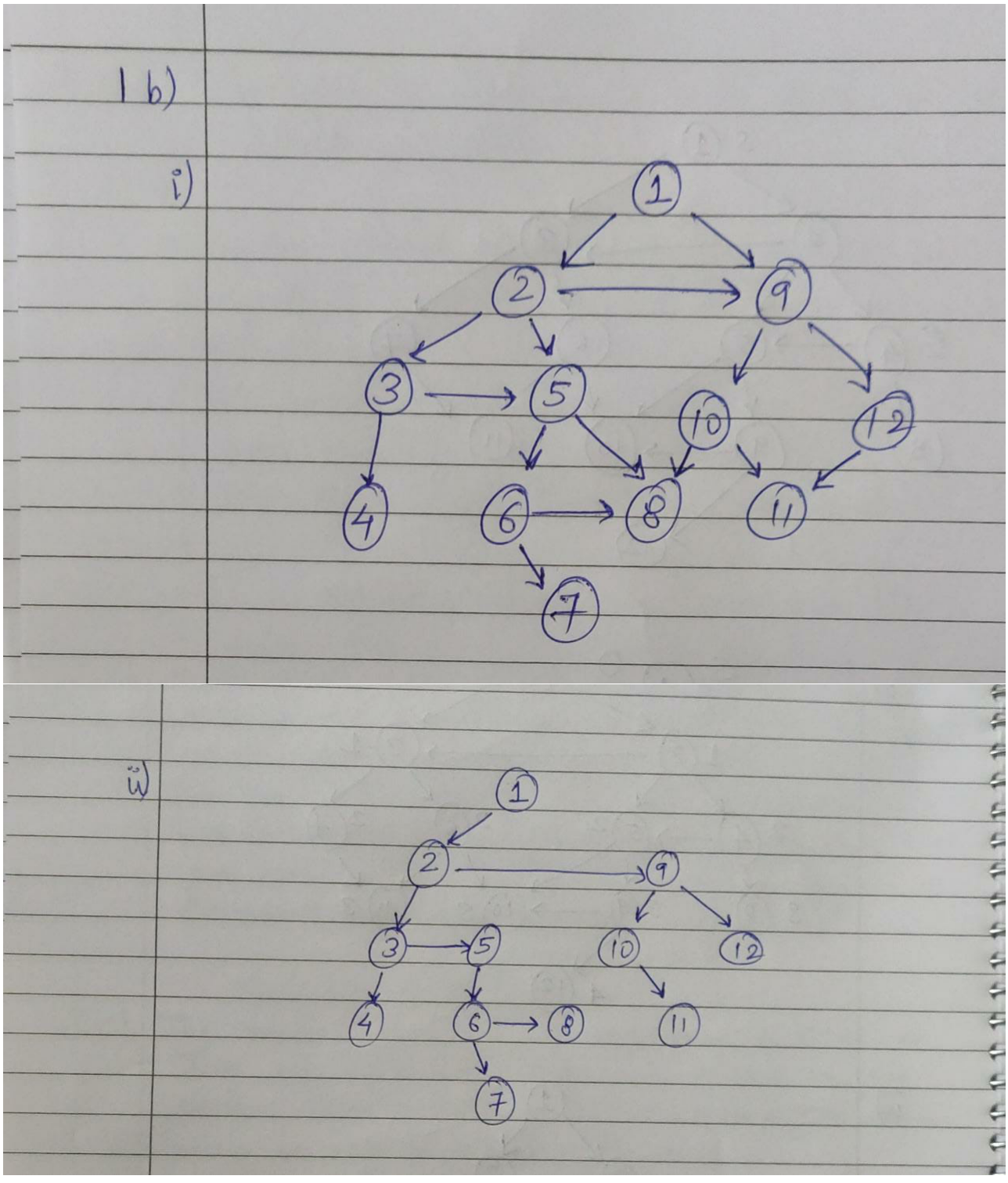






b. [10 points] Perform a depth-first search on the directed graph below (on the left). (i) Number the nodes according to the order in which they are visited at the first time (when they become gray). Show the order numbers inside the circles. (ii) Show the depth-first tree(s).





2. [15 points] Please modify the depth first search algorithm (slide 36 and 37 of the graphs basics lecture notes) to find all connected components in an undirected graph. Comment on where you made the modification. Your modified algorithm needs to print out each component ID (starting from 1) and the corresponding vertices. For example, your output

for the DFS example (slide 38-41 of the graphs basics lecture notes) will look like the following: Component 1: u, v, y, x. Component 2: w, z.

Solution:

```
DFS-Visit(u)
{
   time = time + 1;
   d[u] = time;
   color[u] = gray;
   // Add u to the current component
   componentId[u] = currentComponentId;
   verticesInComponent[currentComponentId].add(u);

   for each v in adj[u] do
   {
      if color[v] == white
      {
         parent[v] = u;
         DFS-Visit(v);
      }
   }

   color[u] = red;
   time = time + 1;
   f[u] = time;
}
```

Modifications made:

- Added a componentId array to keep track of the component ID (starting from 1) for each vertex during DFS traversal.
- Added a verticesInComponent array to store the vertices in each connected component.
- During DFS traversal, when a new vertex u is visited, it is added to the current component using componentId[u] = currentComponentId and verticesInComponent[currentComponentId].add(u).
- When printing out the connected components, you can iterate over the verticesInComponent array and print the vertices for each component along with the corresponding component ID.

3. *[25 points]* Jack plans to drive from city A to city B along a highway. Suppose g_0, g_1, g_2, …, g_k are the gas stations along the highway and are ordered in increasing distance (in miles) from city A, where g_0 is located at the starting place (city A). Let d_i be the distance (in miles) between g_i and g_{i+1} , $i = 0, \ldots, k - 1$, and we assume that these distances are known to Jack. Each time Jack fills gas to his car, he gets a full tank of gas. Jack also knows the number of miles, m, his car can drive with a full tank of gas. Jack's goal is to minimize the number of times he needs to stop at gas stations for his trip. You can assume that Jack starts from g_0 with a full tank.

a) *[8 points]* Design a greedy algorithm to solve the above problem, i.e., to minimize the number of stops for gas.

3)

a) function { minimizeStops (gas_stations, m):

```
currentPosition = gas_stations[0]  // Start position
numStops = 0                       // No. of gas stops.
remainingMiles = m  // Remaining miles with a ~~gas stop~~
                    // full tank.

for i = 1 to length (gas_stations) - 1:
    distanceToNext = gas_stations[i] - currentPosition
    // Distance to next gas station

    if distanceToNext > remainingMiles:
        numStop = numStops + 1  // Fill up gass
        remainingMiles = m      // Reset miles

    currentPosition = gas_stations[i] // Update current
                                      // position
    remainingMiles = remainingMiles - distanceToNext
    // Update remaining miles

return numStops
```

Assumptions

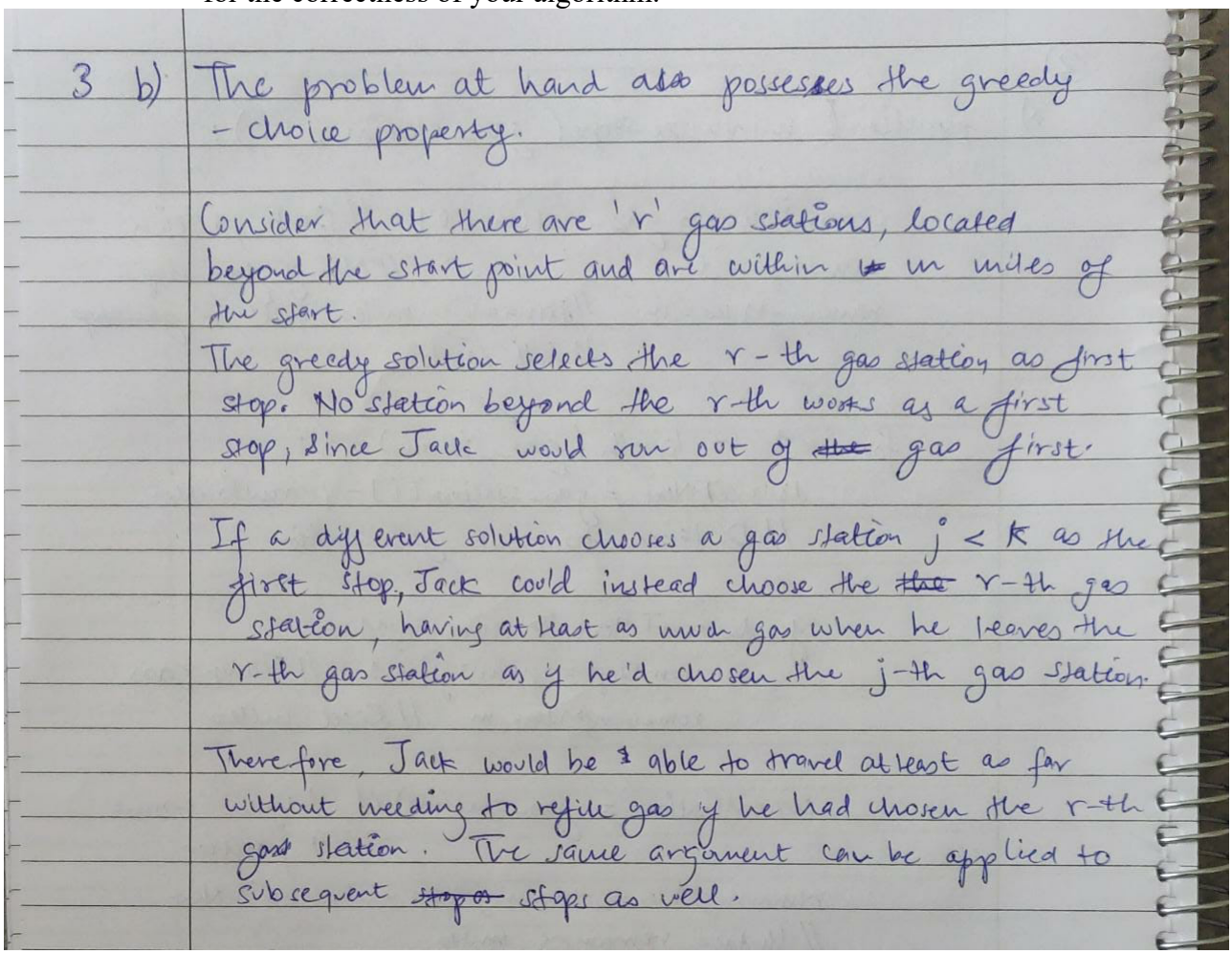
gas_stations are given as an array.

go represented by gas_stations[0].

gas_stations[i] representing i-th gas station from previous gas station

m represents no. of miles Jack's car can drive with full tank.

b) *[12 points]* Show that your greedy algorithm has the *greedy choice property*, i.e., each local decision will lead to the optimal solution. Basically you need to argue for the correctness of your algorithm.



3 b) The problem at hand also possesses the greedy - choice property.

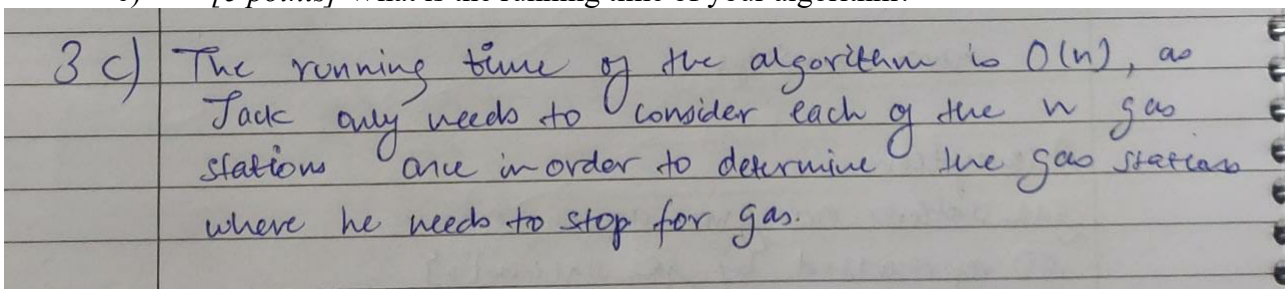
Consider that there are 'r' gas stations, located beyond the start point and are within ~~u~~ m miles of the start.

The greedy solution selects the r - th gas station as first stop. No station beyond the r-th works as a first stop, since Jack would run out of ~~the~~ gas first.

If a different solution chooses a gas station $j < k$ as the first stop, Jack could instead choose the ~~that~~ r-th gas station, having at least as much gas when he leaves the r-th gas station as if he'd chosen the j-th gas station.

Therefore, Jack would be able to travel at least as far without needing to refill gas if he had chosen the r-th gas station. The same argument can be applied to subsequent ~~stops~~ stops as well.

c) *[5 points]* What is the running time of your algorithm?

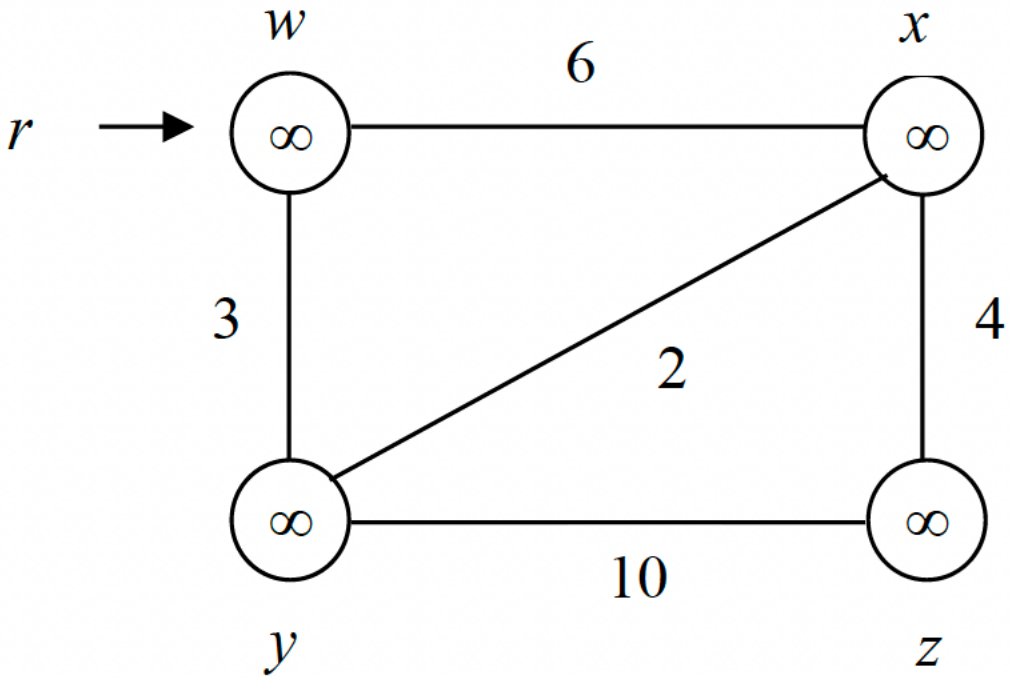


3 c) The running time of the algorithm is O(n), as Jack only needs to consider each of the n gas stations once in order to determine the gas stations where he needs to stop for gas.

4. *[20 points]* Given the Prim's algorithm shown below (a min-priority queue is used in the implementation):

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)        // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

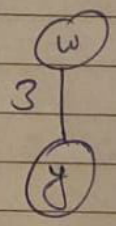
Apply the algorithm to the weighted, connected graph below (the initialization part has been done). Show a new intermediate graph after each vertex is processed in the while loop. For each intermediate graph and the final graph, you need to show the vertex being processed, the new key value for each vertex and edges in the current (partial) MST (draw a directed edge from vertex *v* to *u* if *v*.ρχ = *u*).



4) Lets start with node "w" & find the shortest edge ~~wit~~ weight from node "w"

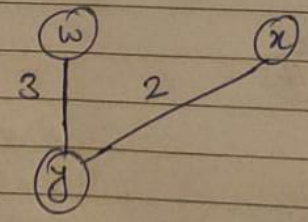
The shortest edge from node "w" is to node "y" with weight = 3

The graph looks as follows.

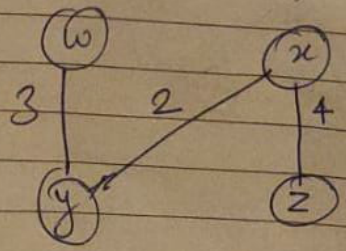


Node "x" which is the neighbor of node "w" is not traversed, but the shortest edge with weight = 2 is picked, which is between node "y" & node "x"

The graph looks as follows.

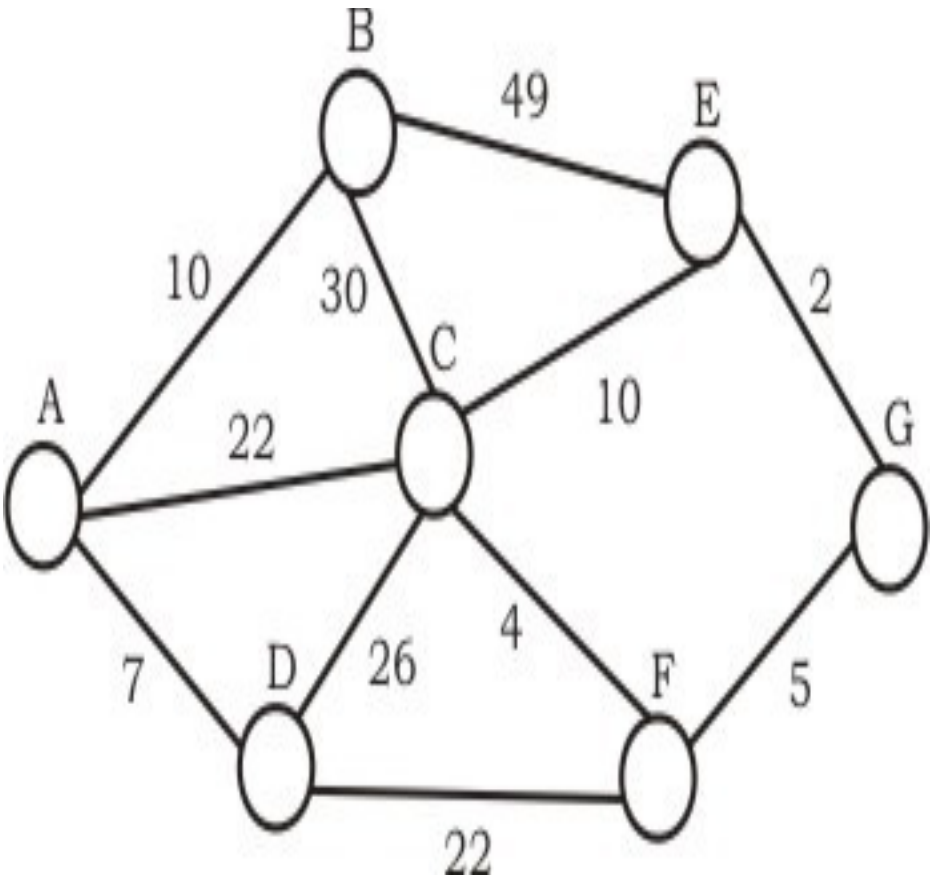


Node "z" will be covered next, the shortest edge is with weight = 4 from node "x" to node "z".
The graph looks as follows.

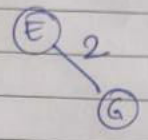


This is the minimum spanning tree.

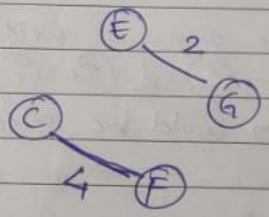
5. *[20 points]* Apply Kruskal's algorithm to the graph below. Show new intermediate graphs with the shaded edges belong to the forest being grown. The algorithm considers each edge in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edges joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.



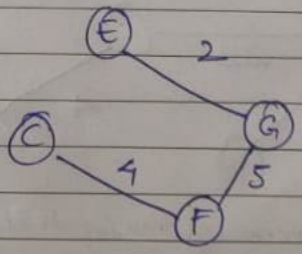
5) - Selecting minimum weight edge (E)—(G) with weight = 2
The intermediate graph is



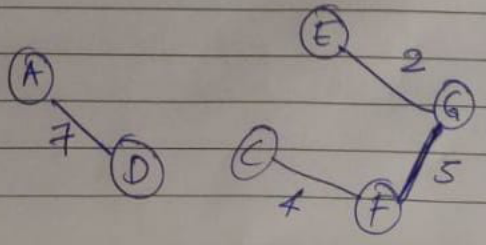
- Selecting minimum weight edge (C)—(F) with ~~we~~ weight = 4
The intermediate graph is



- Selecting minimum weight edge (F)—(G) with weight = 5
The intermediate graph is

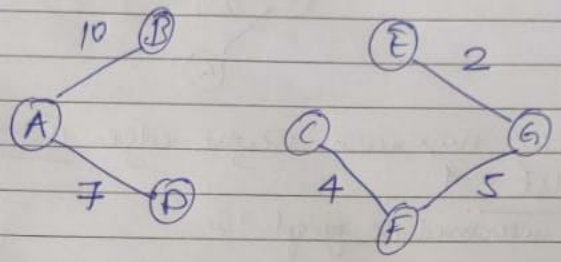


- Selecting minimum weight edge (A)—(D) with weight = 7

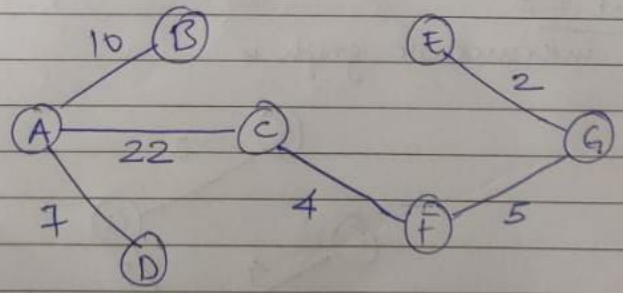


__/__/__

- Selecting the minimum (A)—(B) between (A)—(B) & (C)—(E) as both have weight = 10 but (C)—(E) will form a cycle.



- Selecting the minimum weight edge (A)—(C) with weight = 22
Between (A)—(C) & (B)—(C), (A)—(C) is minimum.
(C)—(E) too could be selected, but will form a ~~cycle~~ cycle.
The graph is:



The minimum spanning tree.