BFS(G,S)

Breadth First Search (BFS).

```
For each vertex $u \in G.V - \{S\}$
    $u.color$ = WHITE
    $u.d = \infty$
    $u.\pi$ = NIL
$S.color$ = GRAY
$s.d = 0$
$S.\pi$ = NIL
$Q = \emptyset$
ENQUEUE(Q, S).
While $Q = \emptyset$
    u = DEQUEUE(Q)
    For each $v \in G.Adj[u]$
        if $v.Color$ == WHITE
            $v.color$ = GRAY
            $v.d = u.d+1$
            $v.\pi = u$
            ENQUEUE(Q, V)
    $u.Color$ = BLACK.
```

u → vertex

source vertex – S

Q – Queue

BFS colors each of the vertices white, gray or black. All the vertices are initialized to white when they are constructed. A white vertex is an undiscovered vertex. When a vertex is initially discovered it is colored gray, and when BFS has completely explored a vertex it is colored black.

This means that once a vertex is colored black, it has no white vertices adjacent to it. A gray node, on the other hand, may have some white vertices adjacent to it, indicating that there are still additional vertices to explore.

DFS(G)

DEPTH FIRST SEARCH (DFS)

```
For each vertex $u \in G.V$
    $u.Color$ = WHITE
    $u.\pi$ = NIL
time = 0.
For each vertex $u \in G.V$
    if $u.color$ = = WHITE
        DFS – VISIT (G, u)
```

DFS – VISIT (G, u)

```
time = time +1    // white vertex u has just been discovered.
$u.d$ = time
$u.Color$ = GRAY
For each $v \in G.Adj[u]$    // explore edge (u,v)
    if $v.color$ = = WHITE
        $v.\pi = u$
        DFS – VISIT (G, V)
$u.Color$ = BLACK    // blacken u; it is finished
time = time +1
$u.f$ = time
```

Big O notation

Worst Case Analysis:- In the worst case analysis) we calculate upper bound on running time of an algorithm.

For Linear Search, the worst case happens when the element to be searched is not present in the array

∴ Complexity of L.S is $O(n)$

Theta notation

Average Case Analysis:- In average case analysis, we take all possible inputs and calculate computing time for all of the inputs.

Sum of all the Calculated values and divide the sum by total no. of inputs.

$$A.C.C = \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$$

$$= \frac{\theta((n+1) * (n+2)/2)}{n+1} = \theta(n).$$

Omega

Best Case Analysis:-

we calculate lower bound on running time of an algorithm.

We must know the case that causes minimum no. of operations to be executed

In the linear search problem, the best case occurs when x is present at the first location.

The no. of operations in worst case is constant (not dependent on n). So time complexity in the best case would be

$\theta(1)$.

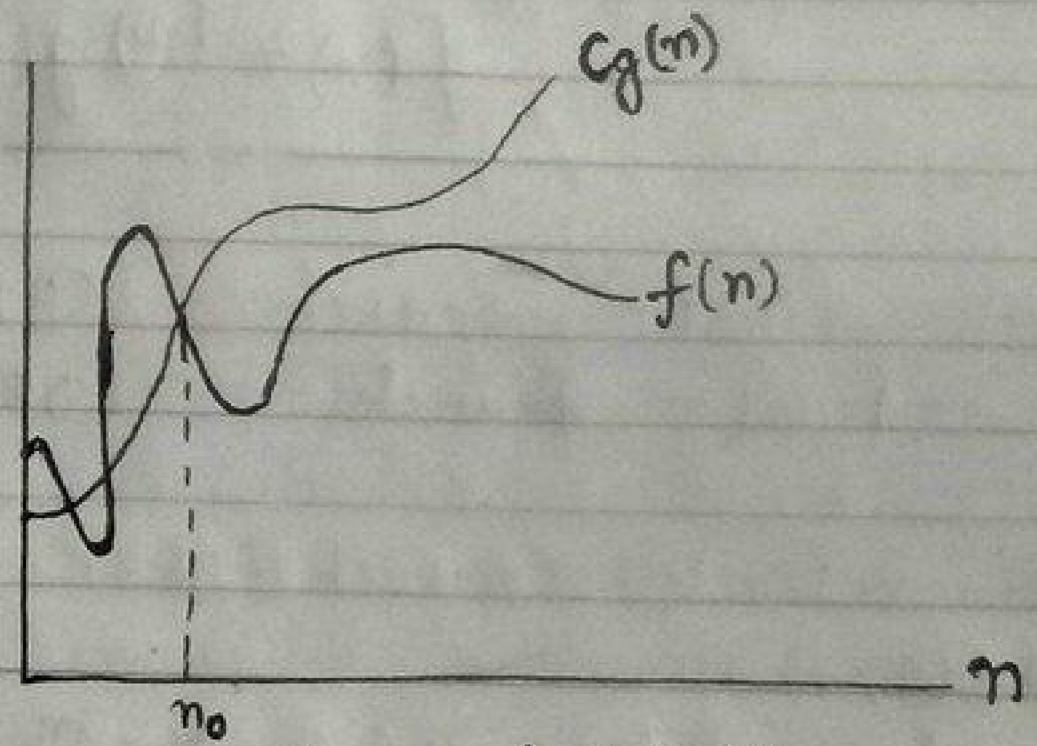
$\theta(n \log n)$

For some algorithms, all the cases are asymptotically same i.e. there are no worst & best cases. for ex:-

Merge Sort:- Merge sort does $\theta(n \log n)$ operations in all case.

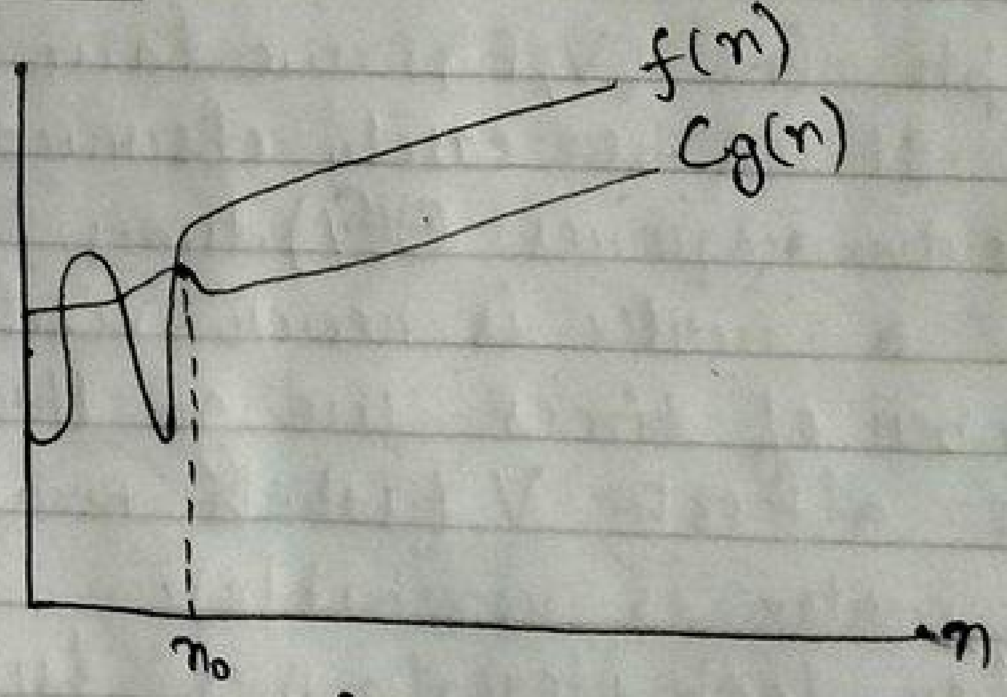
For Insertion sort, the worst case occurs when the array is reverse sorted & the Best case occurs when the array is sorted in the same order as output.

Big O notation:



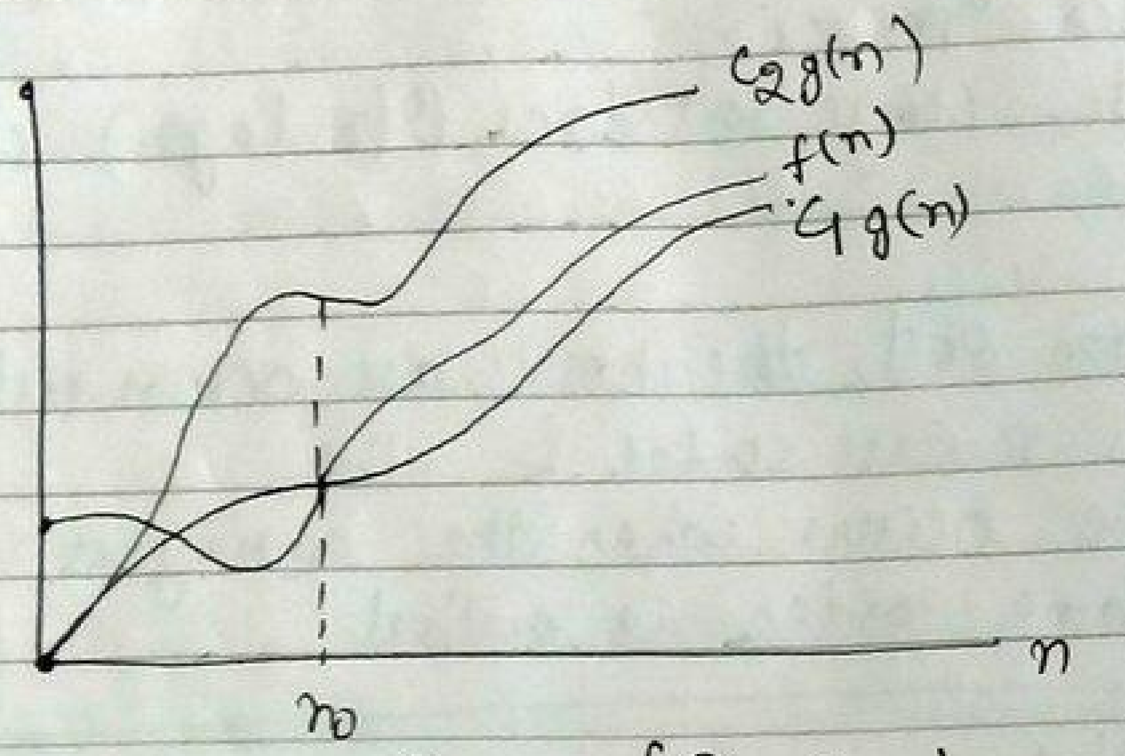
$f(n) = O(cg(n))$

Omega (Ω) notation:-



$f(n) = \Omega(g(n))$.

Theta (Θ) notation :-



$$f(n) = (\Theta\, g(n))$$

* Breadth -First- Search (BFS)
* Depth First- Search (DFS).

* Breadth- First - Search (BFS) :-

```
BFS(G, S)
1) for each vertex u ∈ G, V-{S}.
```

Analysis :→ (BFS) :-

Considering graph $G = (V, E)$, we have $V = |V|$ and $E = |E|$. It can be easily observed that initialization portion requires $\Theta(V)$ times. It is to be noted that a vertex is never visited twice, thus the number of times we go through the while loop is almost V (which means from source each vertex is reachable).

The inner for loop iterations is proportional to $d_G(u)+1$. Here, it is taken because a constant amount of time is needed to set up the loop even if $d_G(u)=0$.

Thus, we have the following running time after summing up all the vertices.

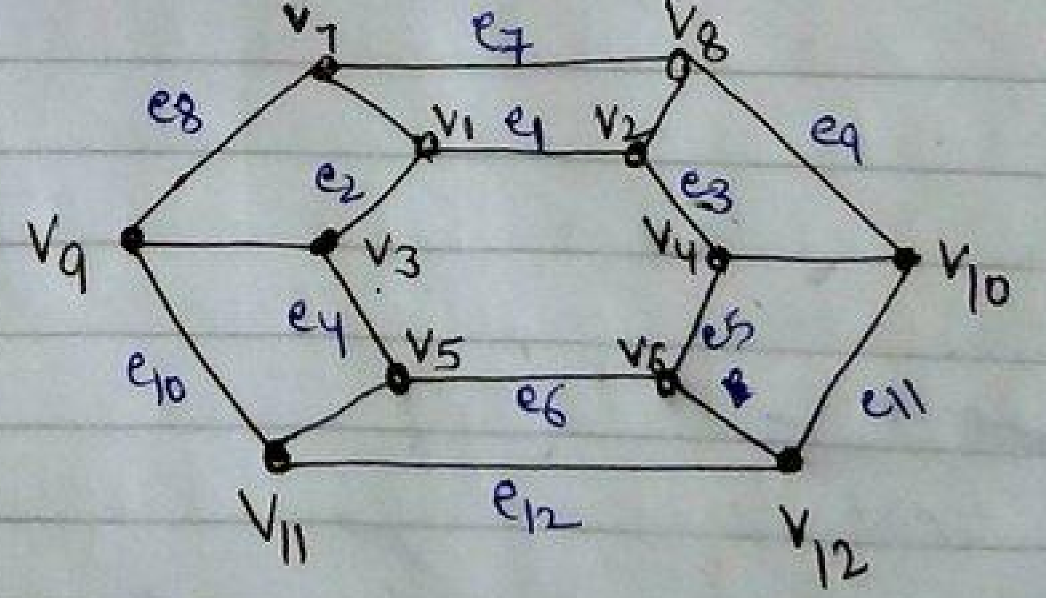
$$T(V) = V + \sum_{u \in V} (d_G(u) + 1)$$
$$= V + \sum_{u \in V} (d_G(u)) + V)$$
$$= 2V + \sum_{u \in V} (d_G(u))$$
$$= 2V + 2E \in \Theta(V + E)$$

For directed graphs the analysis is essentially the same.

Graph :- A Graph $G = (V, E)$ consists of finite non-empty set of objects V, where $V(G) = \{v_1, v_2, v_3 -- v_n\}$ called vertices, and another set E, where $E(G) = \{e_1, e_2, e_3 -- e_m\}$, whose elements are called edges.

From the given set E(G), each edge 'e_k', $1 \le k \le m$ is identified with an unordered pair (v_i, v_j) of vertices. The vertices v_i, v_j from the given set V(G), associated with edge 'e_k' are called end vertices of 'e_k'.

The pictorial representation of graph is shown in fig(i). in which the vertices are represented as points and each edge as a line segment joining its end vertices.



fig(i) Graph with twelve vertices & twelve edges

From fig (i)

$V(G) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}\}$

$E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}\}$

end - vertices for each e_k are

$e_1 = \{v_1, v_2\}$

$e_2 = \{v_1, v_3\}$

$e_3 = \{v_2, v_4\}$

$e_4 = \{v_3, v_5\}$

$e_5 = \{v_4, v_6\}$

$e_6 = \{v_5, v_6\}$

$e_7 = \{v_7, v_8\}$

$e_8 = \{v_7, v_9\}$

$e_9 = \{v_8, v_{10}\}$

$e_{10} = \{v_9, v_{11}\}$

$e_{11} = \{v_{10}, v_{12}\}$

$e_{12} = \{v_{11}, v_{12}\}$

analysis (DFS) : The running time of depth-First-Search algorithm is (V+E). Because of its recursive nature its analysis is somewhat complex than the BFS analysis. We have observed that recurrences are good ways to analysis recursive algorithms, but due to lack of good notion of 'size' it is not true here, ~~that~~ we can attach with each recursive call.

If we ignore the time spent in recursive calls, then procedure DFS ( ) runs in O(V) time.

It can be easily observed that in a search each vertex is visited only once, and thus the call for procedure DFS visit is made exactly once for each vertex. Each one is analyzed individually and after that their running times are ~~added~~ added. If the running time spent in recursive calls are ignored, then each vertex 'u' can be processed in $O(1+d_G^+(u))$ time.

Thus, the total time for the procedure is

$$T(V) = V + \sum_{u \in V} (1 + d_G^+ (u))$$

$$= V + \sum_{u \in V} d_G^+ (u) + V$$

$$= 2V + E$$

$$= \in \Theta (V+E)$$

For undirected graphs similar analysis holds true.

Difference b/w DFS and BFS.

Depth First Search and Breadth First Search differ in their exploring philosophies:

→ DFS always longs to see what's over the next hill (where pastures might be greener), whereas

→ BFS visits the immediate neighbourhood thoroughly before moving on.

After visiting a node, BFS explores this node (visits all neighbors of the node that have not already been visited) before moving on.

DFS immediately moves on to an unvisited neighbor, if one exists, after visiting a node.

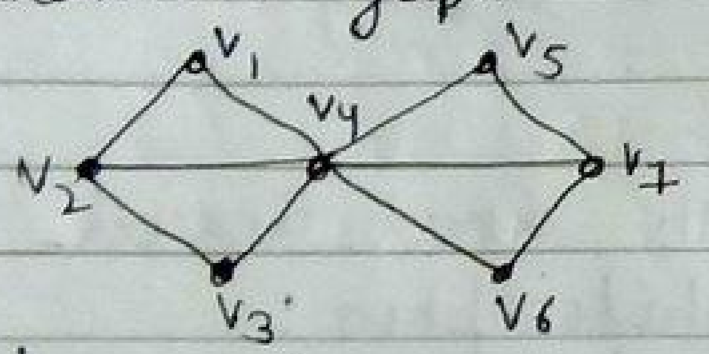
Whenever a DFS is at an explored node, it "backtracks" until an unexplored node is encountered and then continues. This backtracking often return to the same node many times before it is explored.

Minimum Spanning Tree

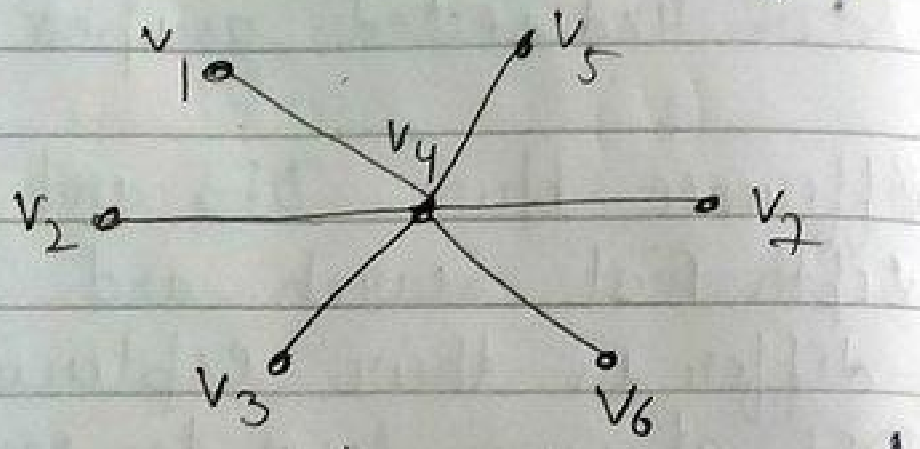
Spanning tree:- Let a graph G = (V, E), if 'T' is a sub-graph of G and contains all the vertices but no cycles/ circuit, then 'T' is said to be a spanning tree.

We are ~~assuming~~ using spanning tree in relation with connected graph.

Undirected graph



Undirected graph
7 vertices
10 edges



spanning tree.
7 vertices
6 edges

Minimum Spanning tree:- If a weighted graph is considered, then the weight of the spanning tree (T) of graph 'G' can be calculated by summing all the individual weights in the spanning tree T. But we have seen that for any graph 'G' there can be many spanning trees. This is also in the case of weighted graphs from which we can get different spanning trees having different weights. A minimum spanning tree (MST) is a spanning tree of minimum weight.

Generic Technique:- For computing the minimum cost spanning tree we are presenting two algorithms (Kruskal's and Prim's algorithm) which are based on Greedy technique.

We have studied that greedy algorithm searches for the choice which gives the optimal solution at each stage repeatedly among all options. The most crucial point about greedy algorithm is that once a choice is made, then by no way that choice is unmade

The partial solution built by Kruskal algo are forests and the partial solution built by Prim's algo are trees.

* **Kruskal algorithm:** In kruskal algorithm; an edge is selected in such a manner that it contains a minimum weight and upon adding to 'M' does not induce any cycle. That means every time the lightest edge is selected first and then is added to 'M' (only if no cycle is obtained).

* **Prim's algorithm:-** The prim's algorithm differs from the kruskal algorithm only in the way of selecting the next safe edge which does not produce cycle upon adding.

The importance of prim's algorithm is that it looks very similar to an algorithm (greedy algorithm) which is known as Dijkstras algorithm, used for finding the shortest paths.

```
MST_ KRUSKALS (G)
A = $\phi$
for each vertex V ∈ G.V
    MAKESET (V)
Sort the edges of G.E into non decreasing
order by weight ω.
    Non- decreasing order by ω.
    If FIND-SET (u) ≠ FIND SET (V)
        A = A ∪ {(u,V)}.
        UNION (u,V).
```

Shorteste Path Algorithms:

1) ~~Sh~~

```
MST_ PRIM (G, ω, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
    r.key = 0.
    Q = G.V
    while Q ≠ $\phi$
    u = Extract_ MIN (Q)
    for each V ∈ G.Adj [u]
    If V ∈ Q and ω(u,V) < V.key
        V.π = u
    V.key = ω (u,V).
```

Some definitions:-

Trees:- A tree is a non- linear data structure, and widely used in many application.

A hierarchical relationship can be best explained with the help of 'Trees'.

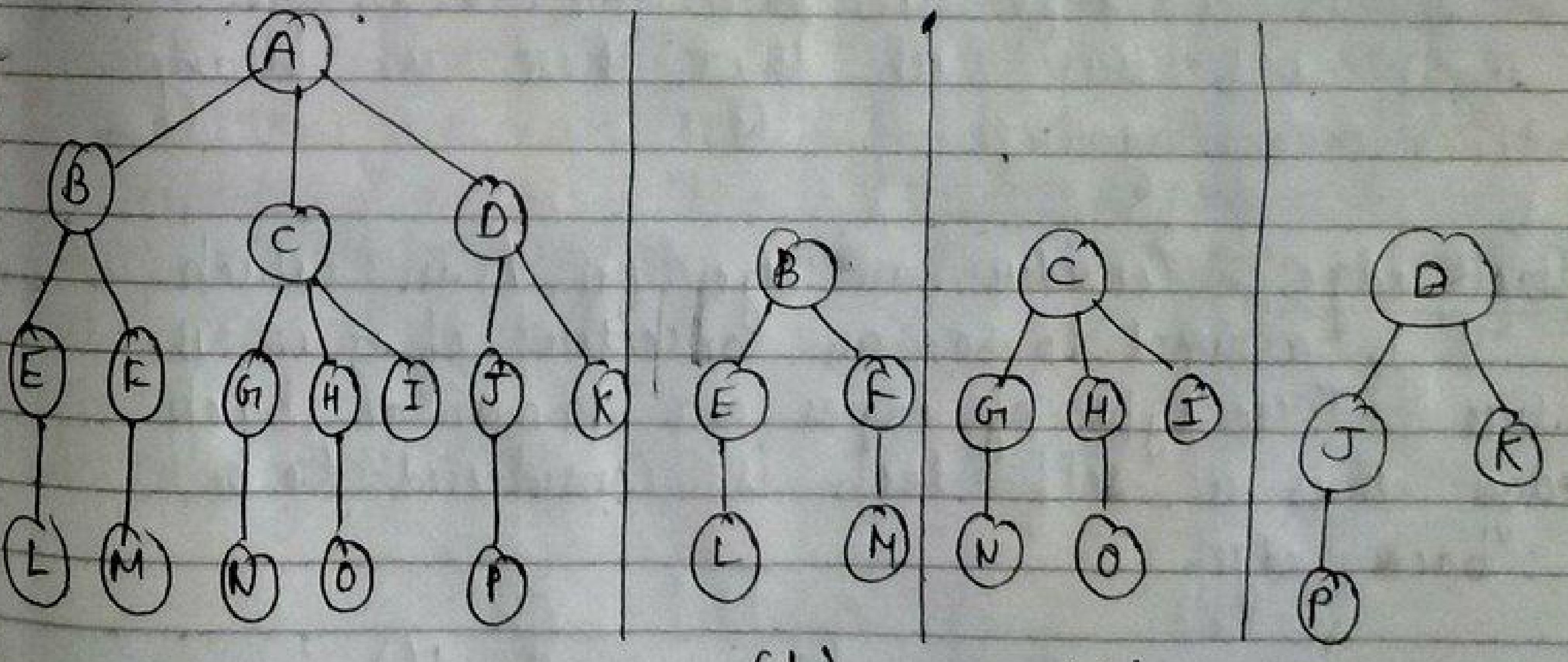
A tree 'T' can be defined as a finite set of one or more nodes, in such a manner that:

- there exits a unique node known as root node.
- the remaining nodes of a tree are partitioned into n≥ disjoint sets - T_1, T_2, T_3 --- T_n, where each of these sets belongs to a tree.

The disjoint sets T_1, T_2, T_3 --- T_n are called the sub- trees of a tree.

The definition of a tree is recursive as its sub- trees are 'tree' once again.

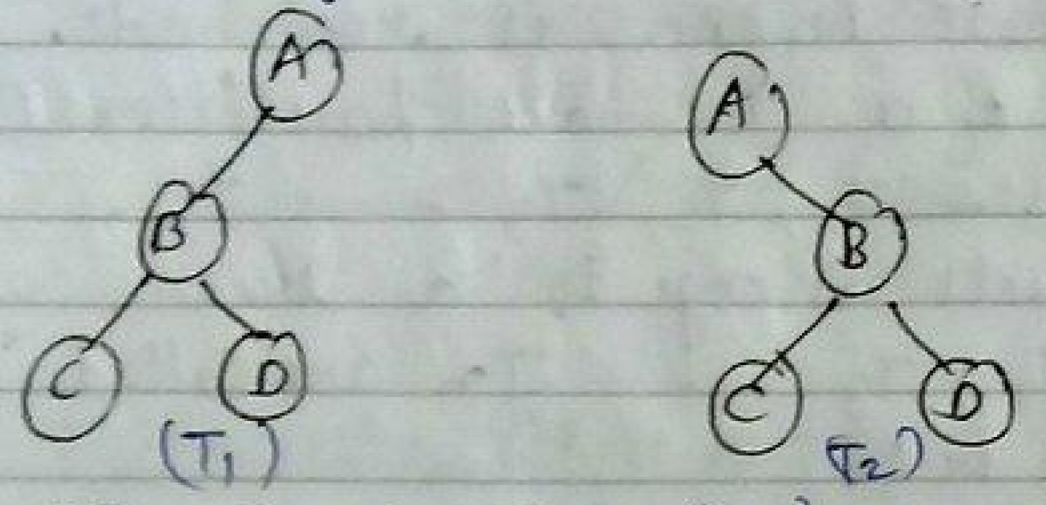
* Forest:- A forest can be defined as a set of n≥0 disjoint trees. A forest can be generated if we remove a root node from the given tree.



(a) Tree (b) (c) (d)

Fig (ii) Tree, b, c, d represent a forest.

* **Binary tree :-** Binary tree is a special class of data structure in which the number of children of any node is restricted to atmost two. In addition to this, the children of a node in a binary tree are ordered. Thus, we distinguish the left sub-tree (child) and the right sub-tree (child), but in case of a General tree the order is irrelevant.



The Binary tree 'BT' may also have zero node, and can be defined recursively as

- An empty tree is a binary tree
- A distinguished node (unique node) known as root node.
- The remaining nodes are divided into two disjoint sets 'L' and 'R' where 'L' is a left sub-tree and 'R' is a right sub-tree such that these are binary trees once again.

Back edge :- Consider an edge (u, V) in which 'V' is assumed to be an ancestor of u in the tree. These type of edges are known as back edge. Thus a self-loop is considered to be back edge.

Forward edges :- Consider an edge (u, V) in which 'V' is a proper descendant of 'u' in the tree. These types of edges are knownas forward edge

Cross edges :- Consider an edge (u, V) in which u and V are not ancestors or descendents of one another. Such type of edges are known as cross edges. In this the edge may go b/w different trees of the forest.

Debugging :- Debugging is the process of finding and correcting the cause at variance with the desired and observed behaviours. Acc to E. Dijkstra, debugging can only point to the presence of errors but not to their absence

* **Notations for algorithm :-**
We have expressed all the algorithms by Pseudocode. Here Pseudocode means that the algorithms that are presented are language and machine independent.
The prefix Pseudo~~means~~ is used to give the information that the code is not meant to be compiled and executed on a computer. It is easy to understand an algorithm by using pseudocode. The pseudocode hides the implementation details and thus one can solely focus on the computational aspects of an algorithm. Pseudocode consists of keywords and English-like Phrase which specify the flow of control.

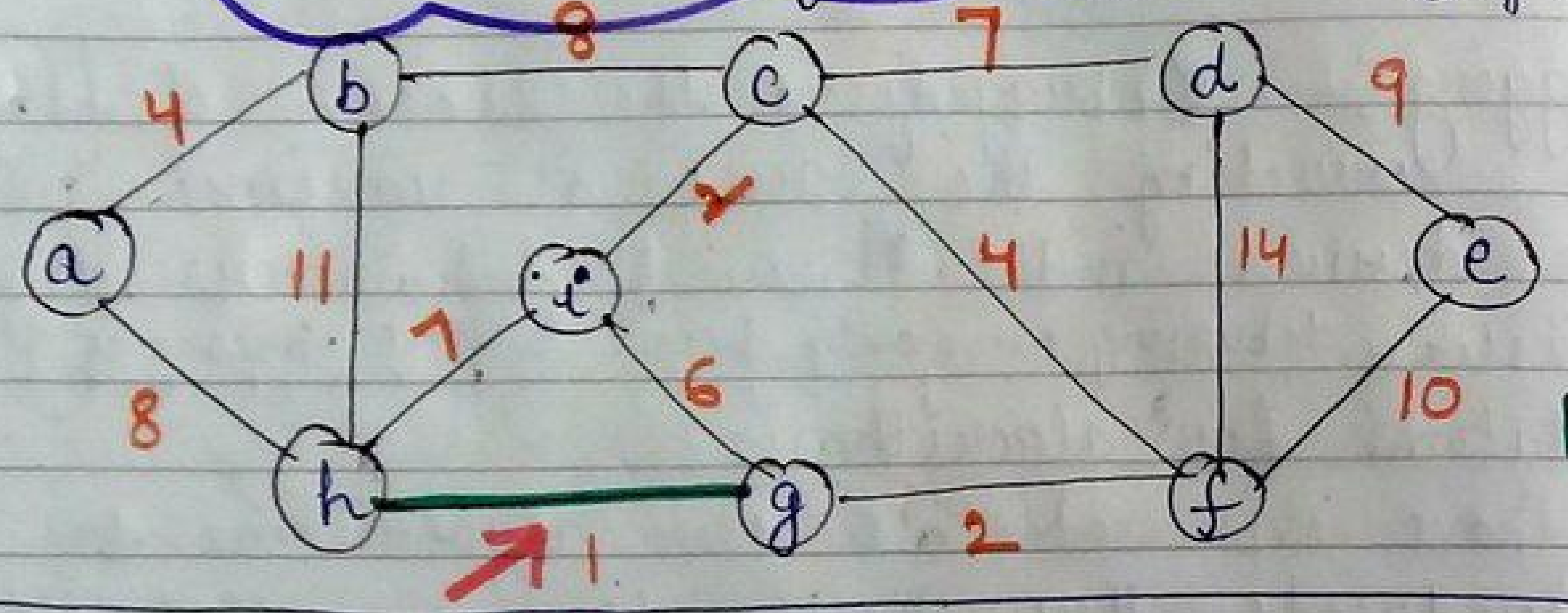
* Topological sort :

h-g	gf	ic	ab	cg	ig	cd	ih	bc	ah	de	ef	bh	df
1	2	2	4	4	6	7	7	8	8	9	10	11	14

Kruskal algorithm

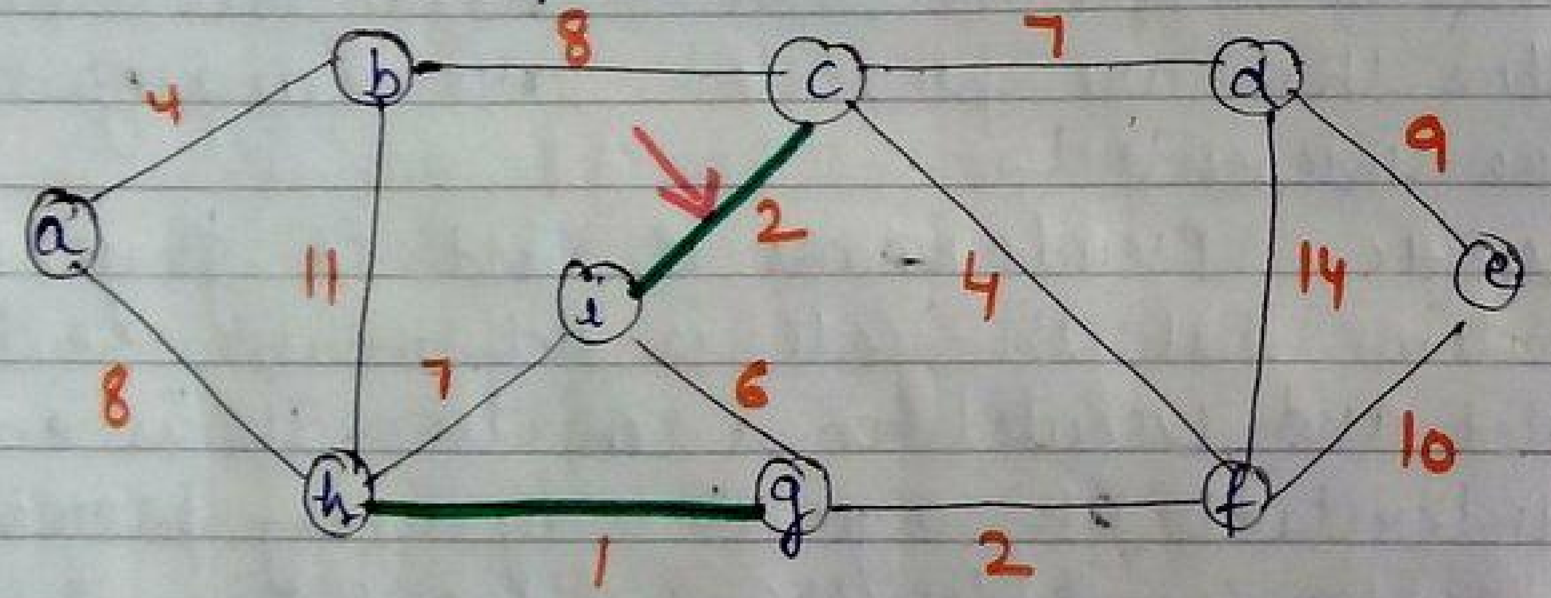
nodes=9
edges=14

(a)
1



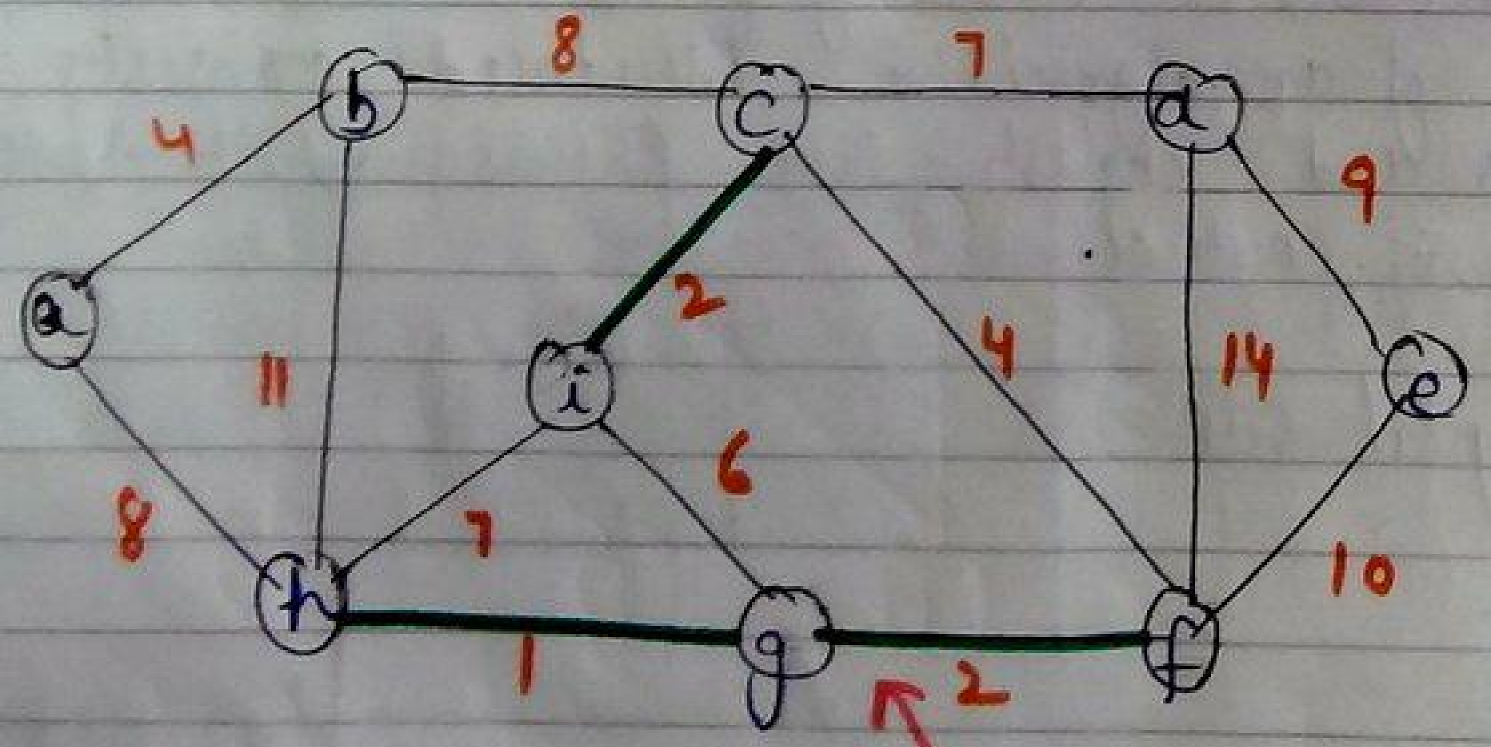
hg

(b)
2



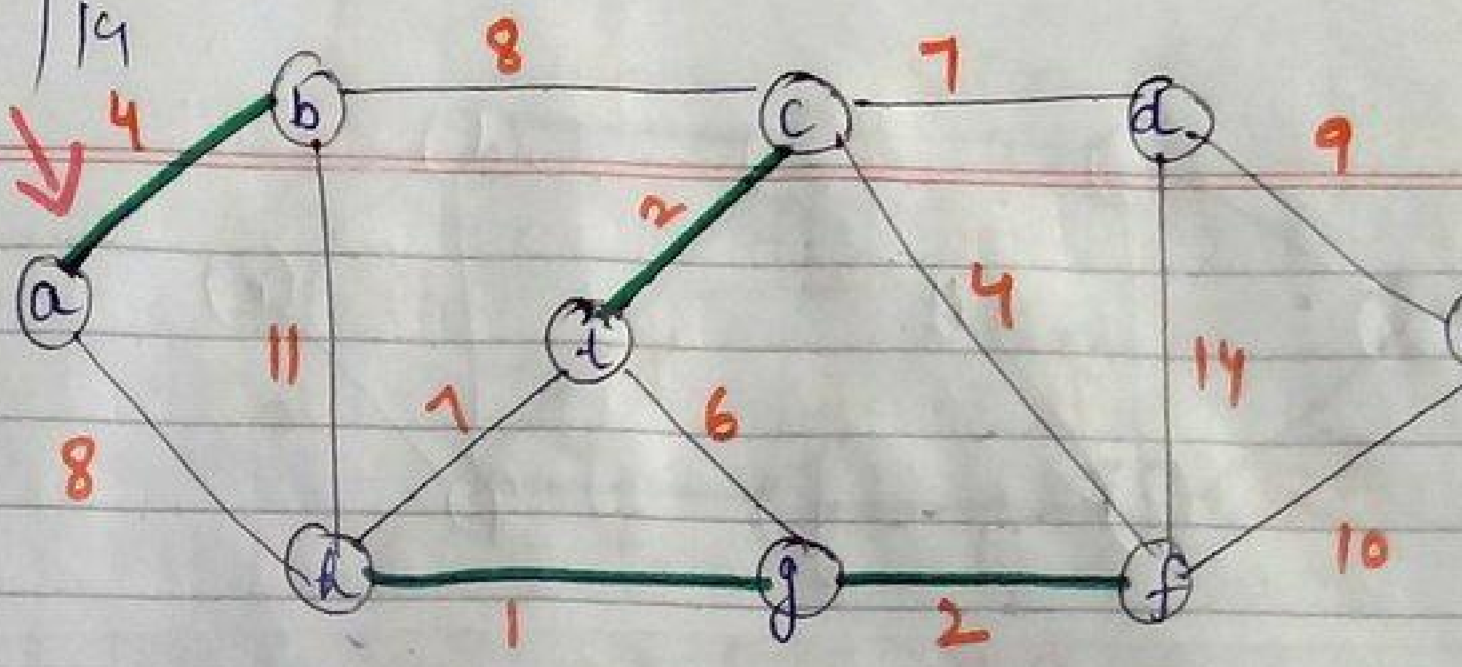
ic

(c)
3



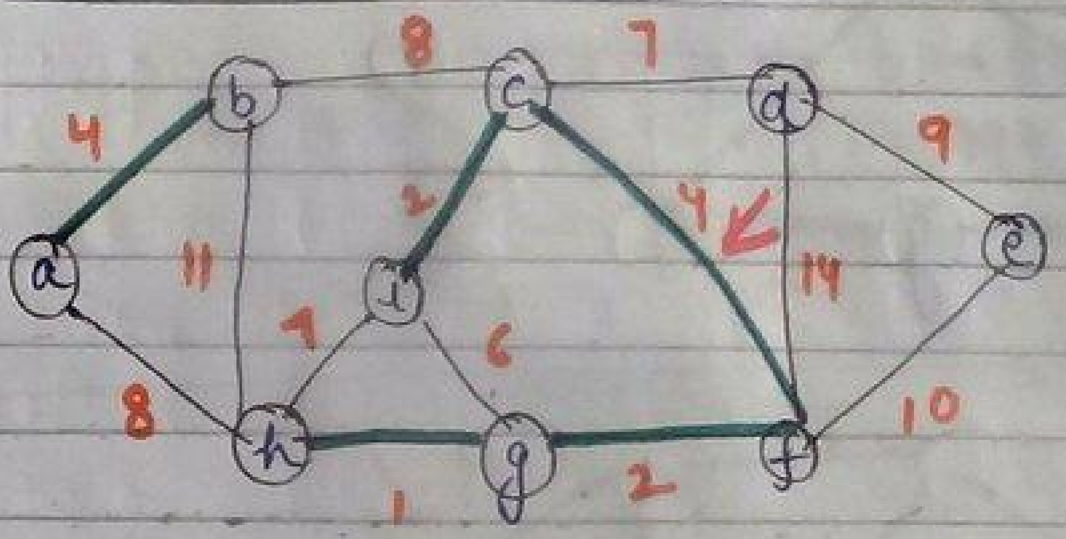
gf

4 d)



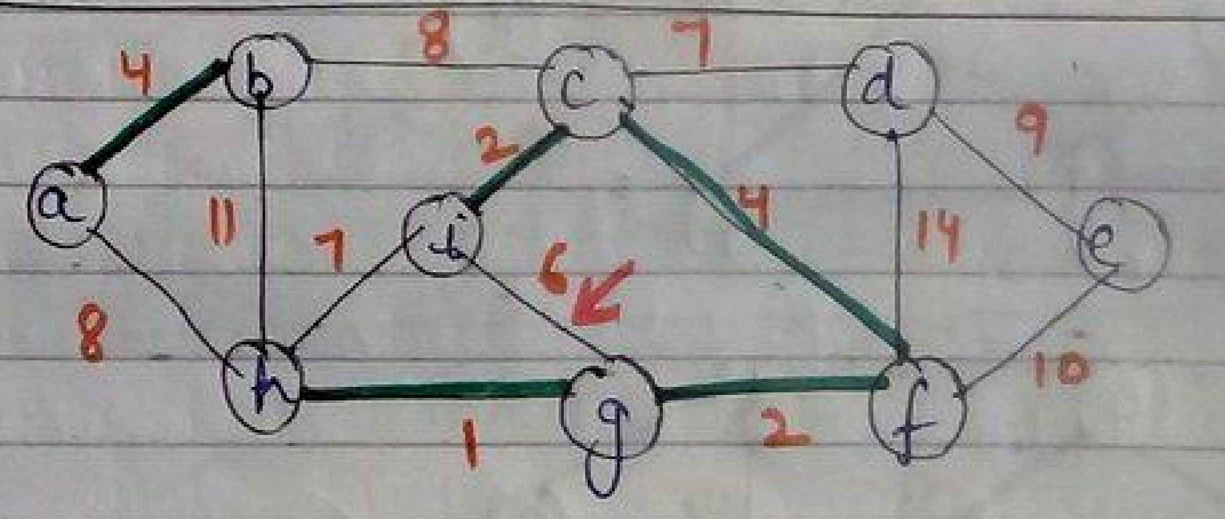
ab

5 e)



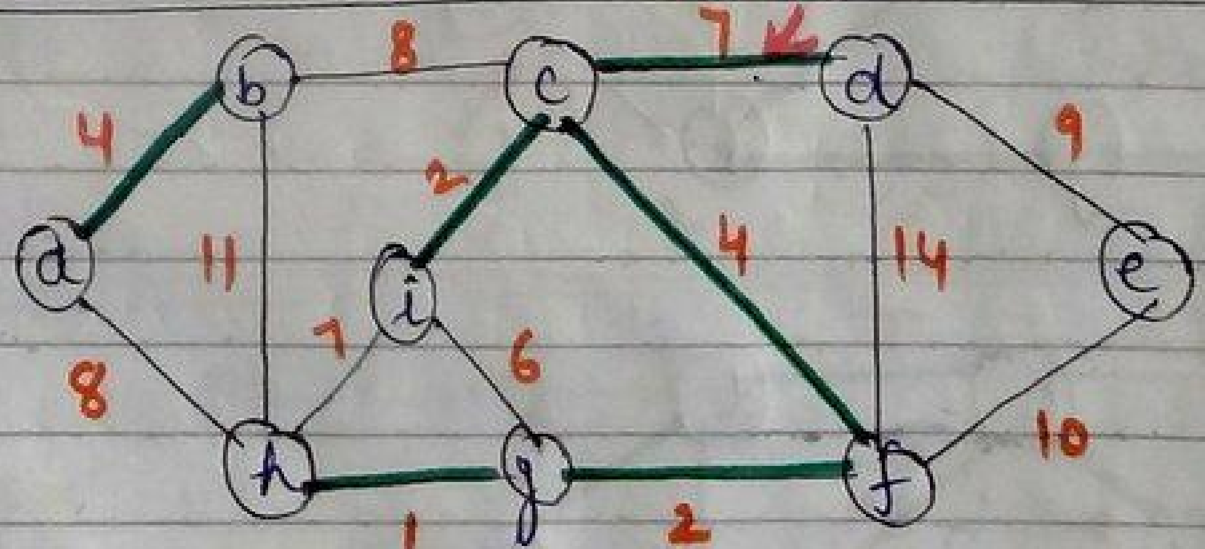
cf

f) 6



ig (x)

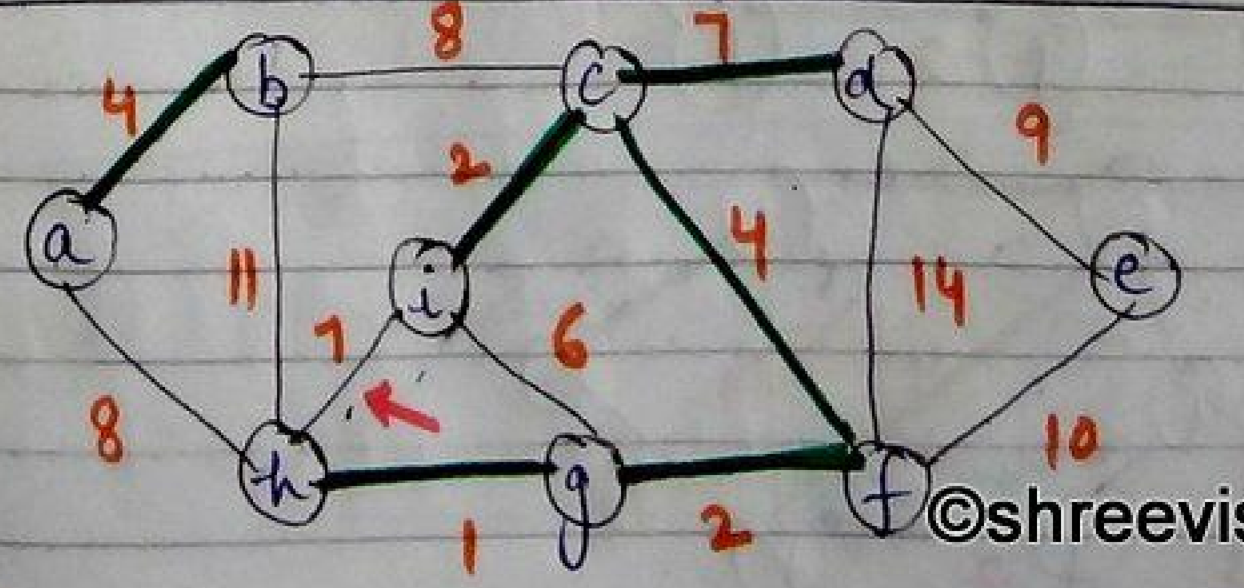
7 g)



~~cd~~

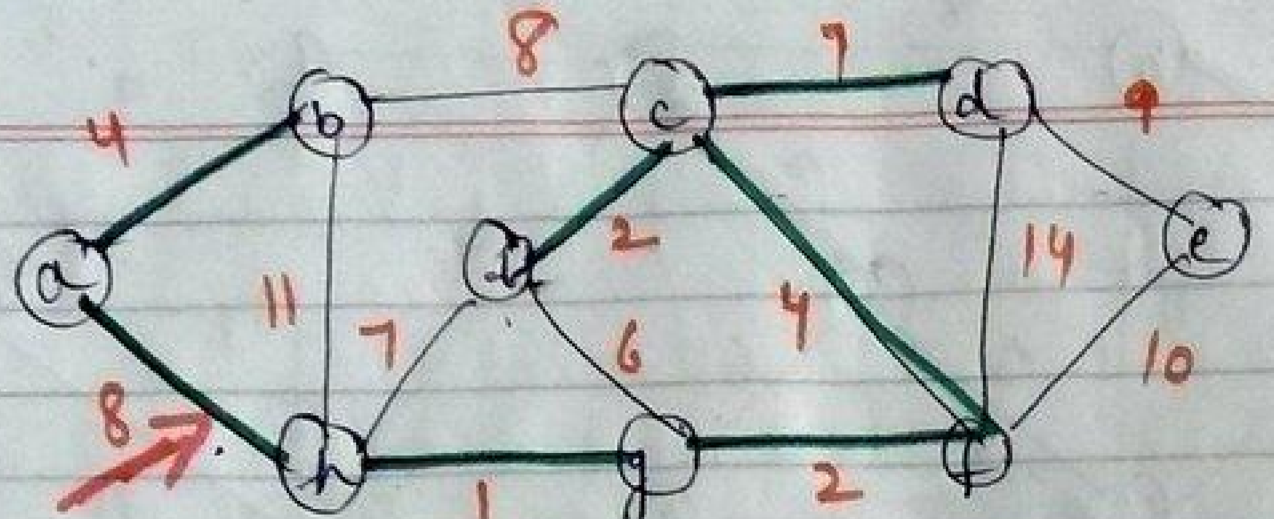
~~bc (x)~~

8 h)

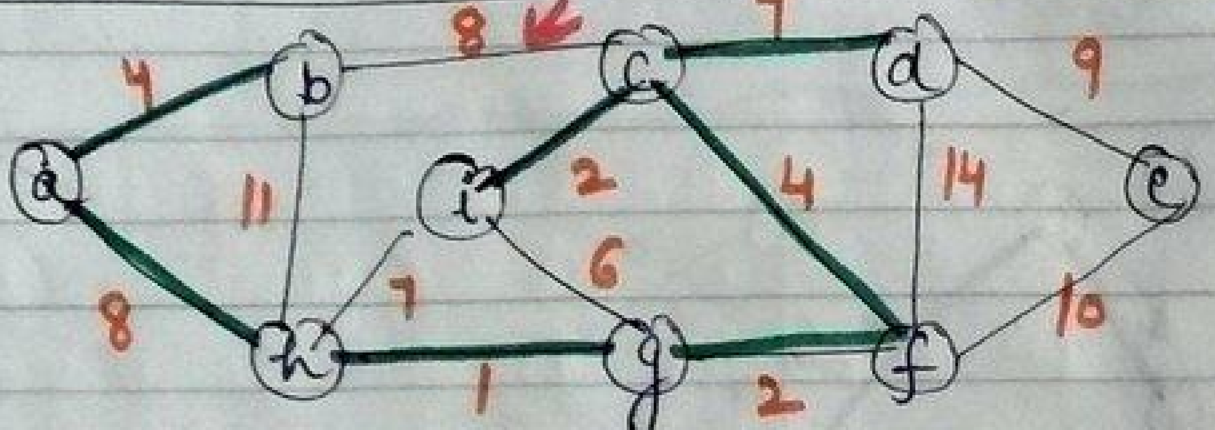


hi(x)

9 i)

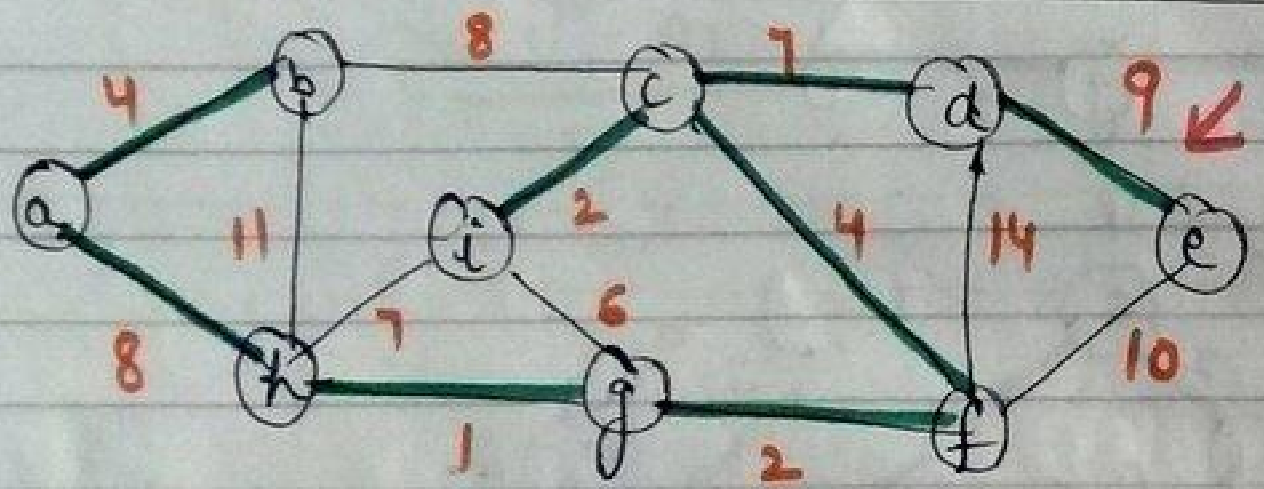


10 j)

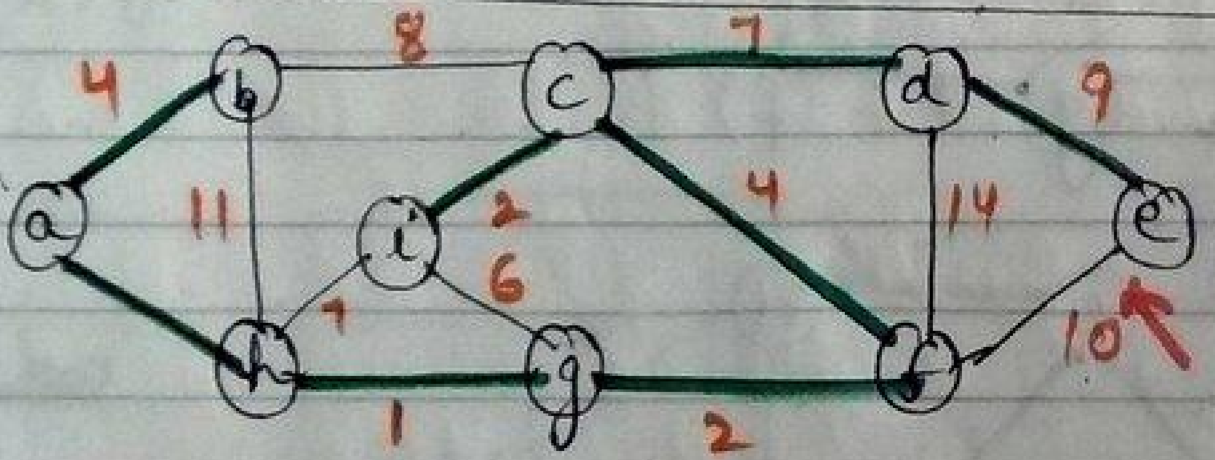


(x)

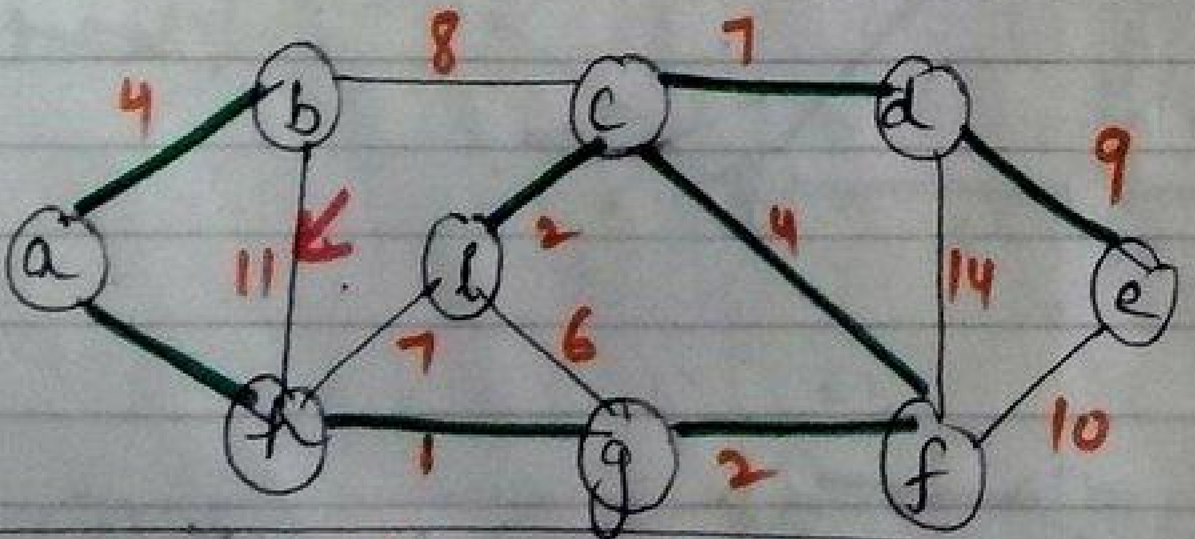
11 k)



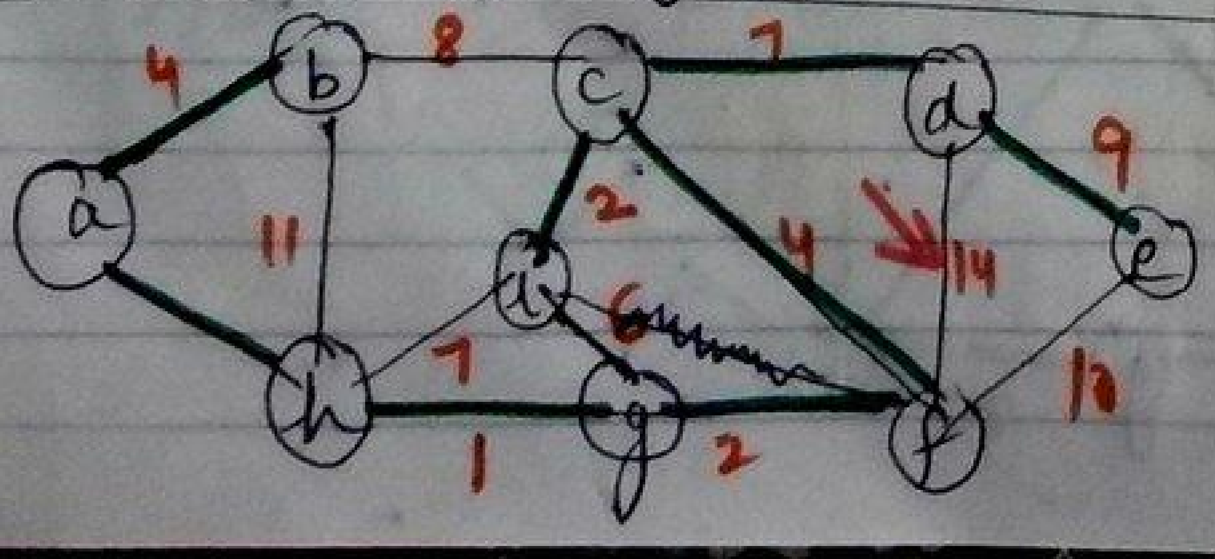
12 l)



13 m)

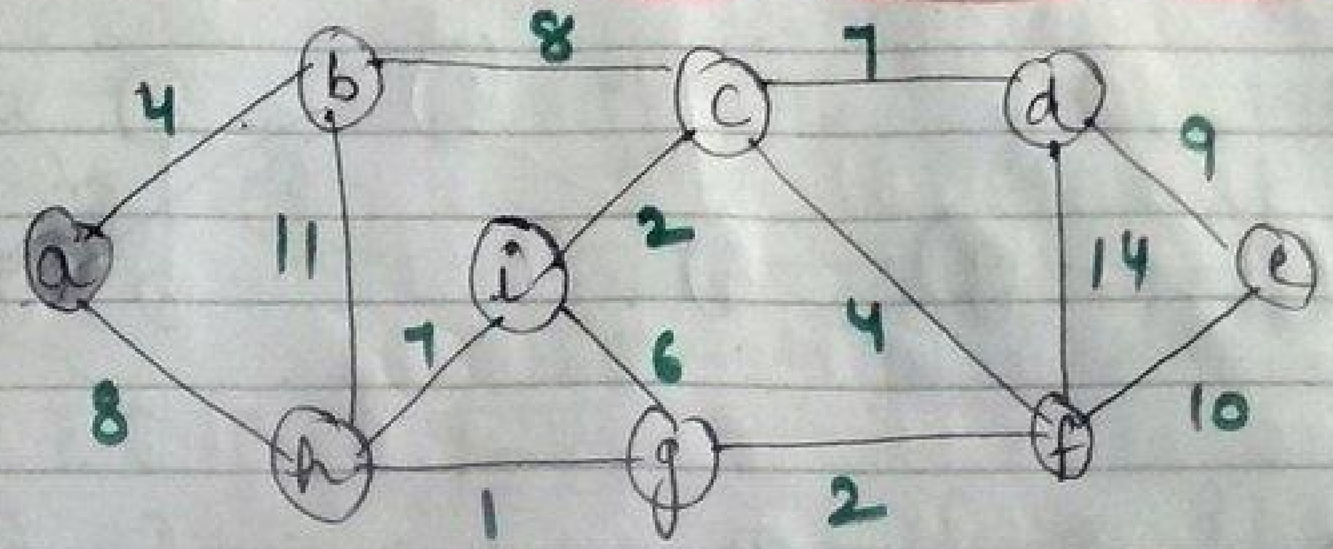


14 n)

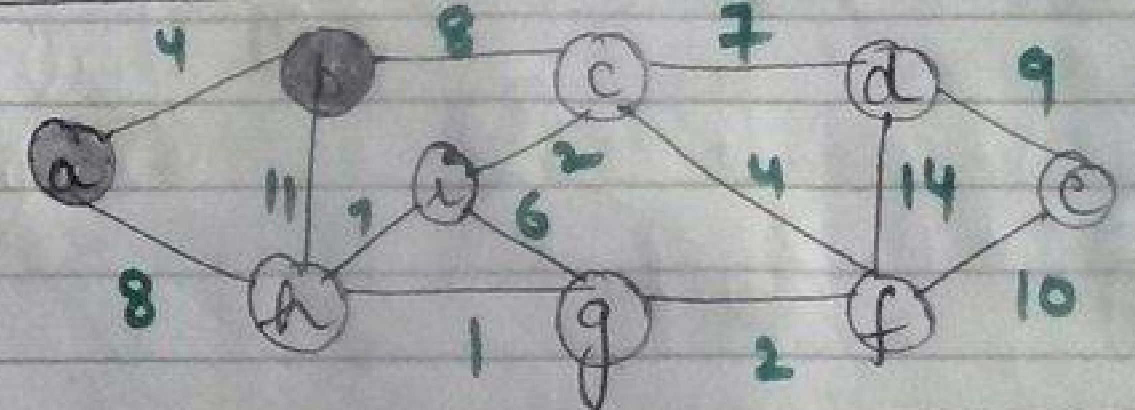


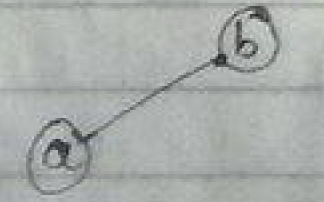
Prim's algorithm

1 a)

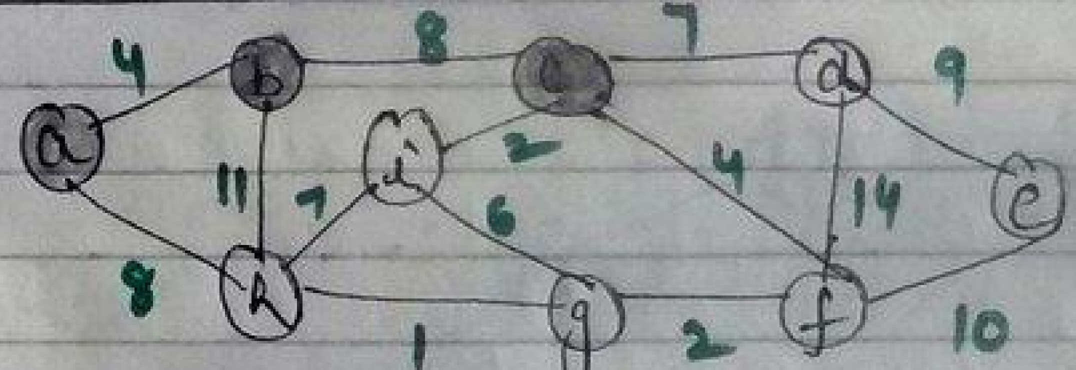


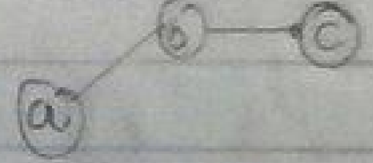
2 b)



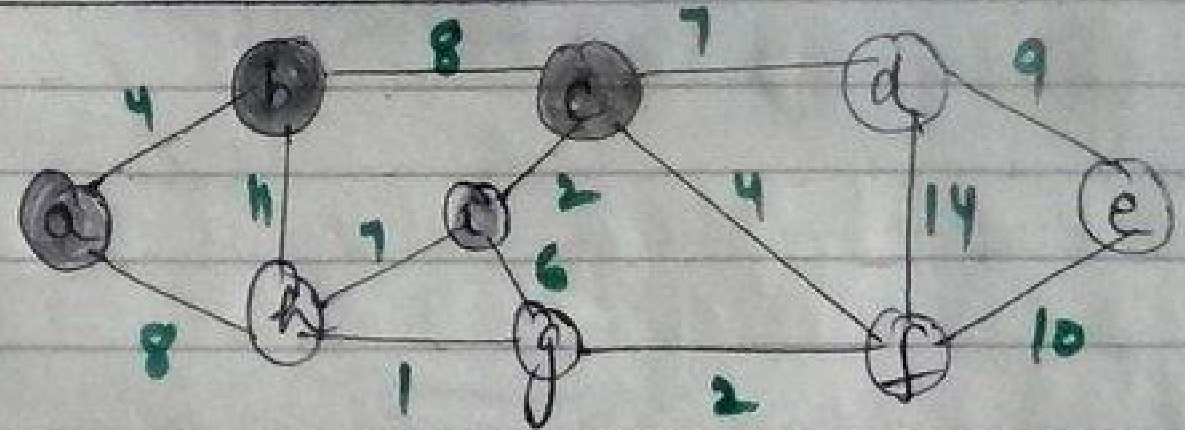


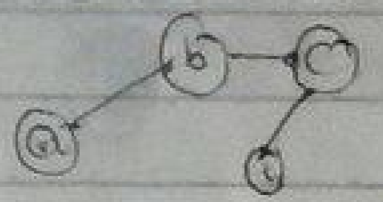
3 c)



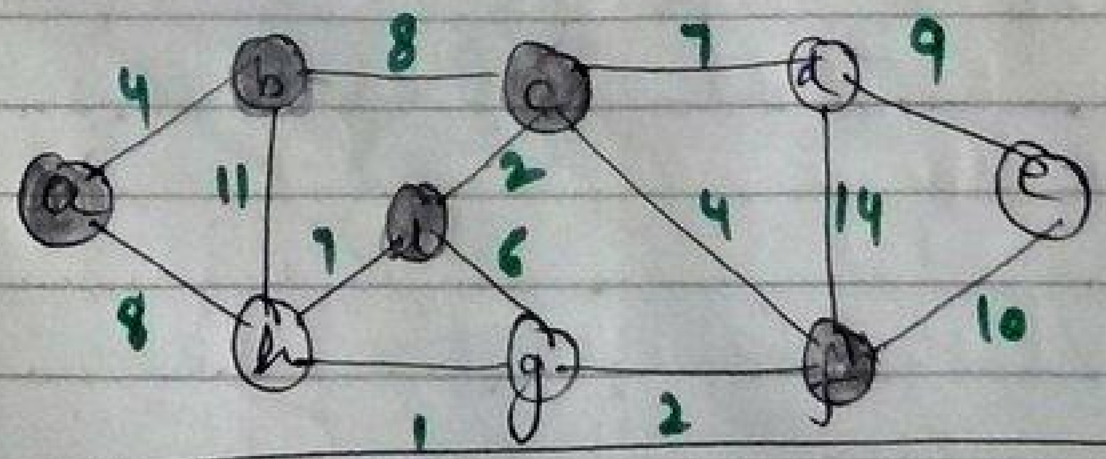


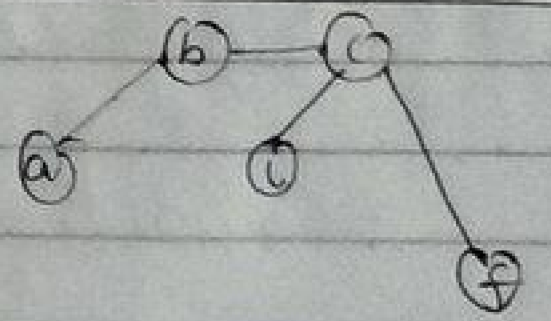
d) 4



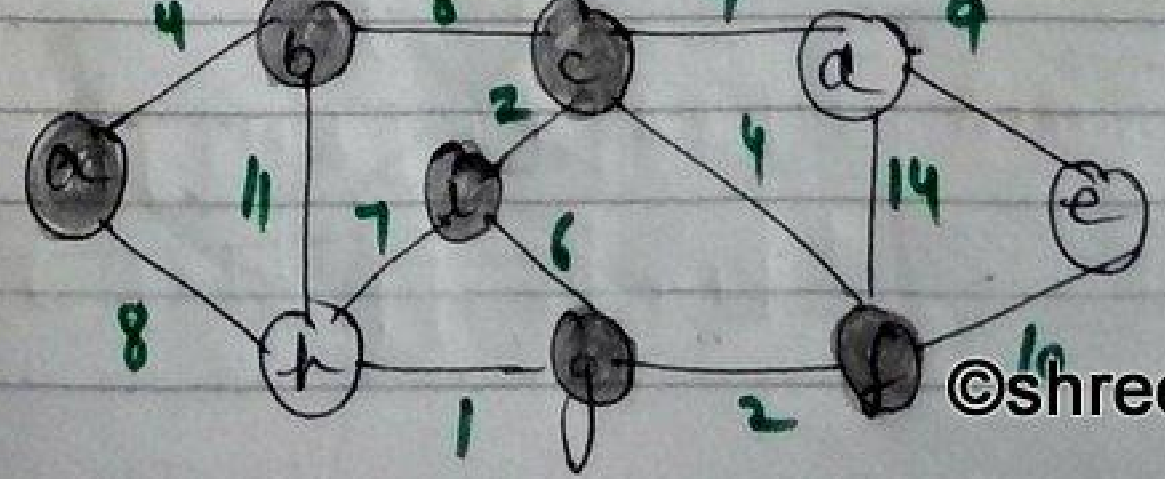


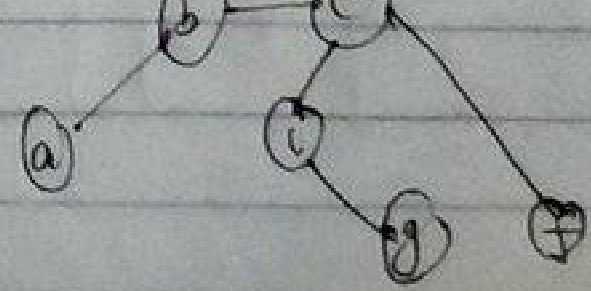
e) 5





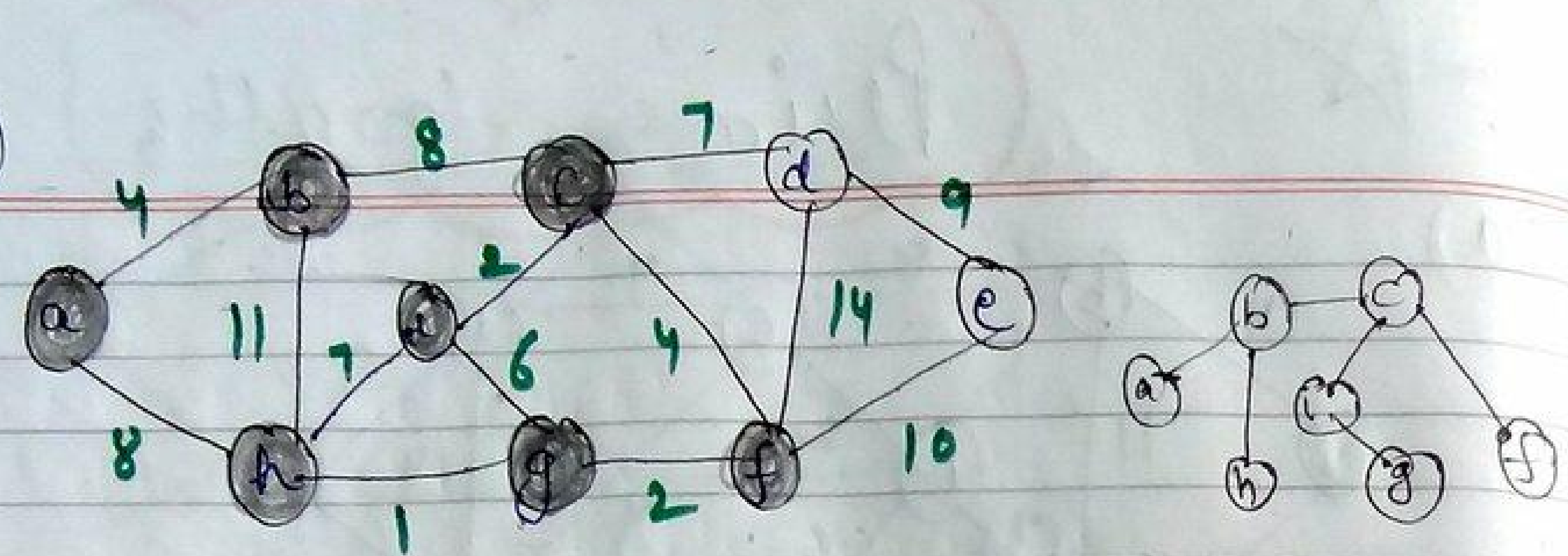
f) 6





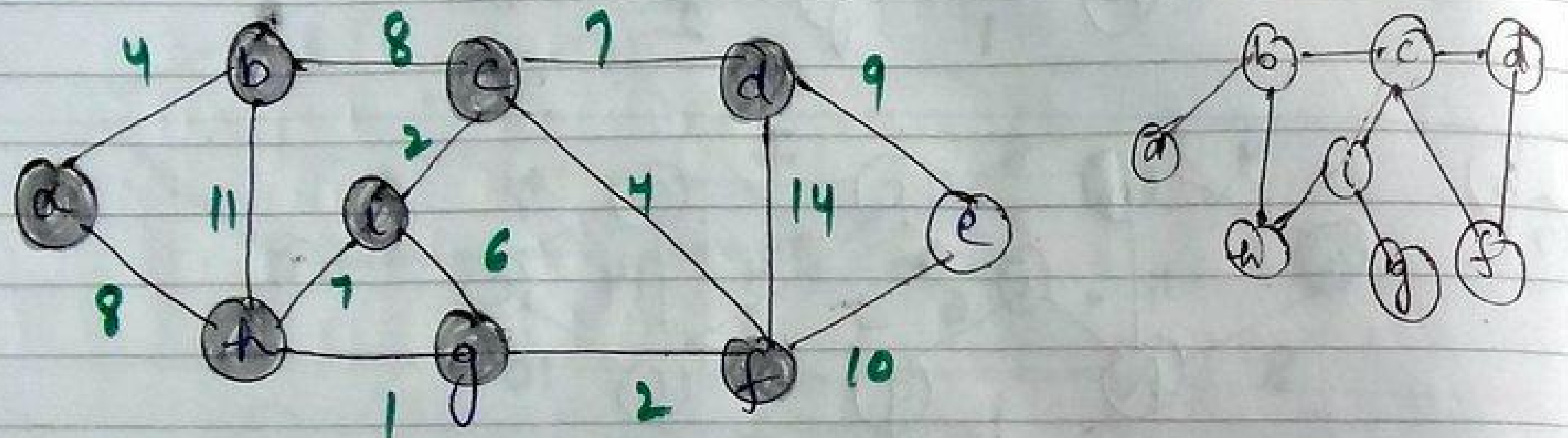
7

g)



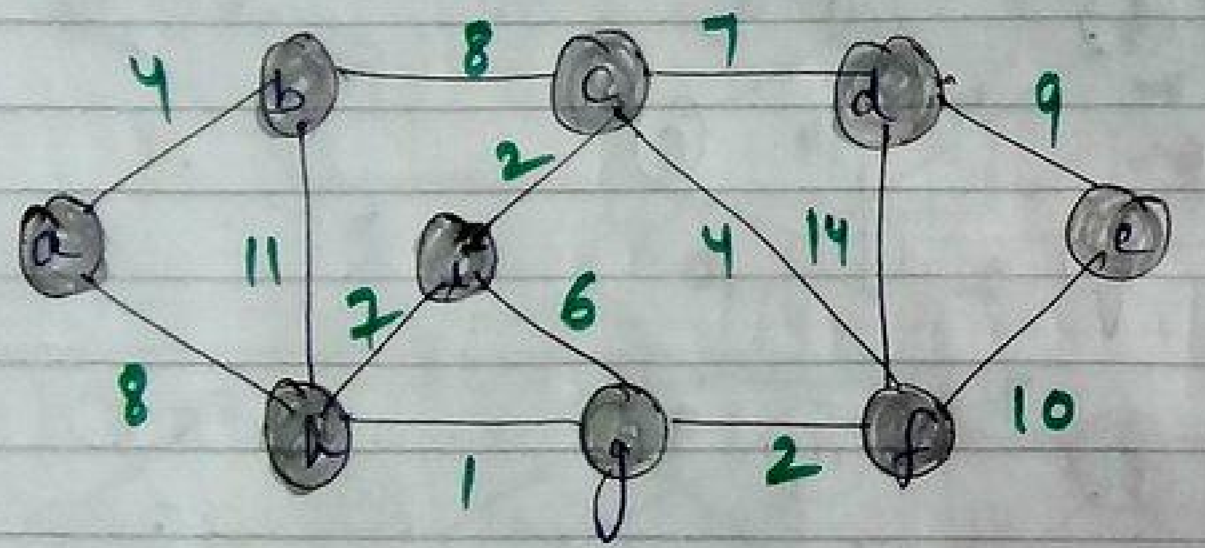
8

h)



9

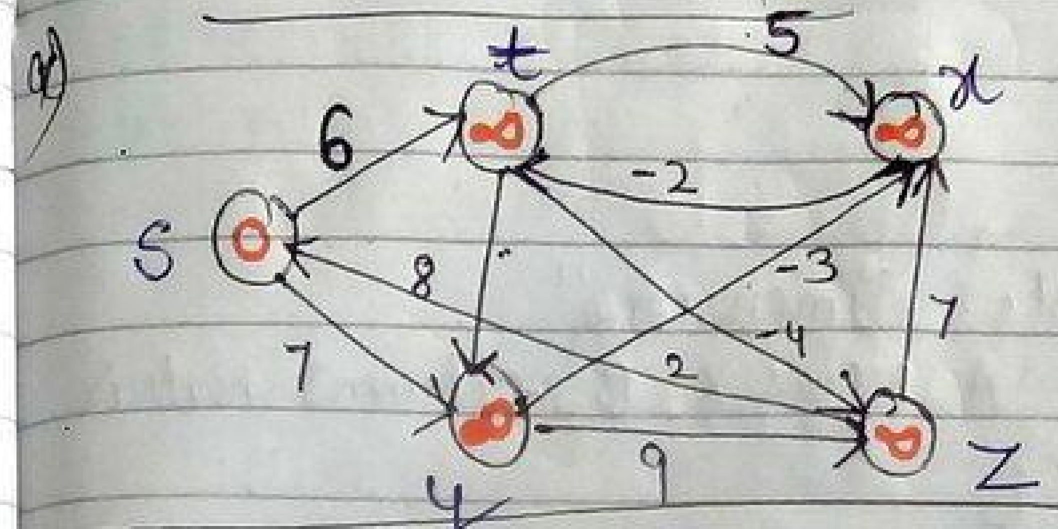
i)



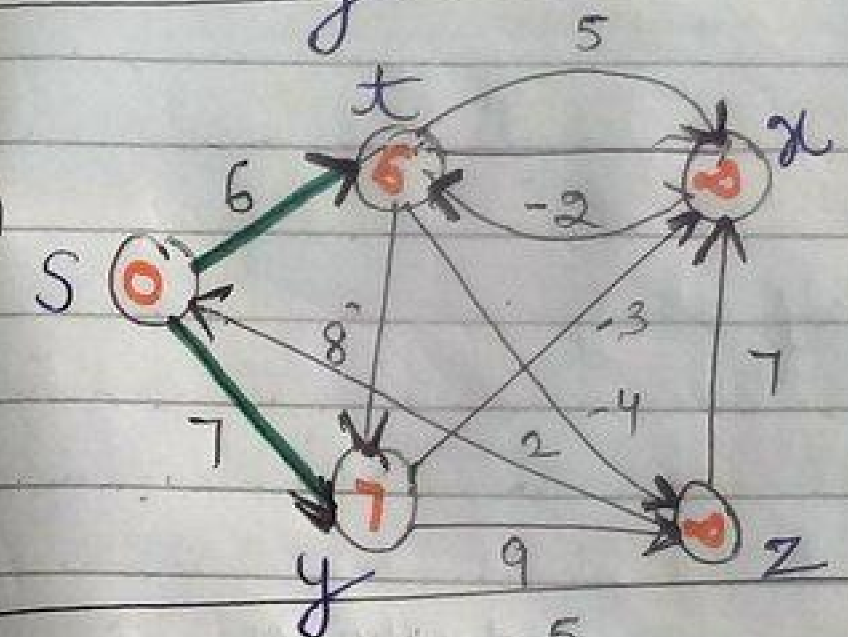
Bellman ford
Dijktra.

Bellman - ford

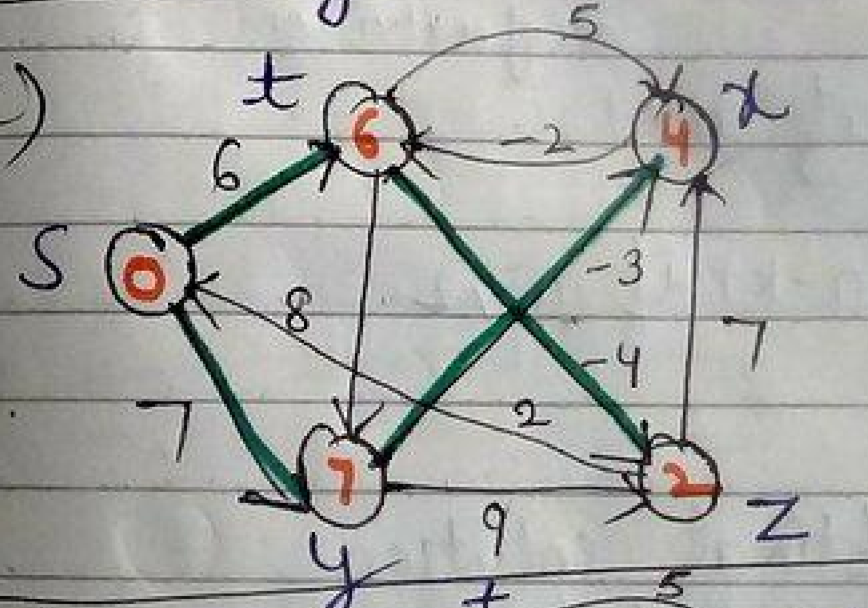
a)



b)

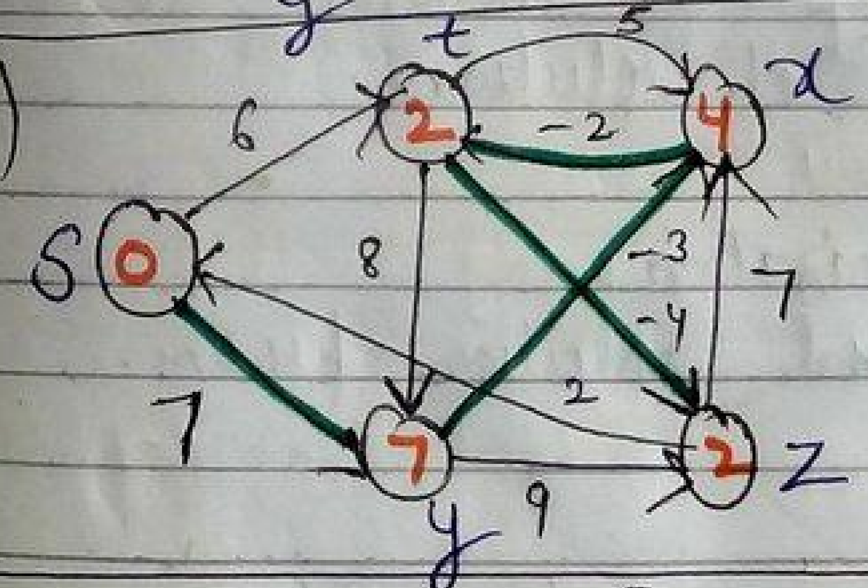


c)

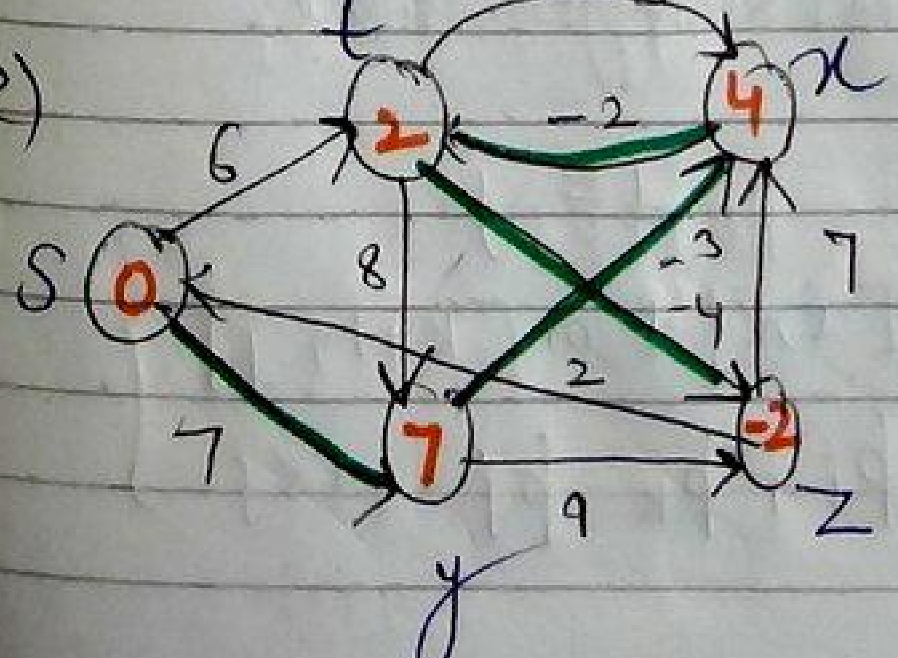


$5 + 7 + -3$
2

d)



e)



t
x
s
y
z
0
2
4
7
-2
6
5
-2
8
-3
-4
7
2
7
9