Answer to the question no: 1

① In a directed graph, adjacency is assymmetric.

↳ Node A is a neighbor of node B only if there is an edge $A \to B$

↳ Node B is not not be a neighbor of Node A unless there is an edge $B \to A$.

Thus, A & B both be called adjacent unless both edges $A \to B$ & $B \to A$.

③ In a directed graph without loops and multiple edges, each vertex ~~cannot~~ can connect to n-1 others vertices, and the direction matters.

$$\text{Max edges} = n(n-1)$$

② An undirected graph without loops and multiple edges is called a simple ~~path~~ graph. The maximum number of edges is the number of unique pairs of n vertices.

$$\text{Max edges} = \frac{n(n-1)}{2}$$

④ To avoid cycles, the graph must be a collection of trees or a single tree. A tree with n vertices has n-1 edges. Thus, the maximum number of edges in a acyclic undirected graph is,

$$\text{Max edges without cycle} = n-1$$

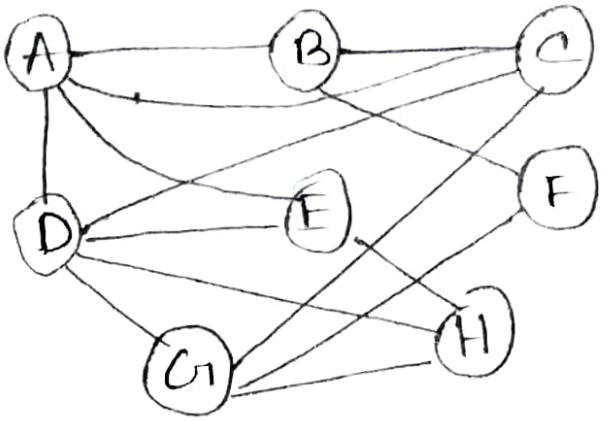
(c) Analyzing the efficiency of using adjacency list over adjacency matrix of different types of graph:

Graph type	Adjacency list	Adjacency Matrix
Undirected	Space efficient for sparse-graphs: $O(V+E)$	Space heavy: $O(V^2)$
directed	space: $O(V+E)$	space: $O(V^2)$
weighted	Easy to include weights in edges	Needs extra space to store weights
Unweighted	Simpler Storage	Takes more space
Sparse	very efficient: only stores existing edges.	Highly ineffient, most of the matrix is unused
Dense	May require more space than the matrix.	Efficient for space, all connections stored

Therefore, we can use adjacency list for sparsed graphs. And we can use adjacency matrix for ~~desed~~ dense graph or any algorithm requiring frequent edges lookups

Answere to the question no: 2

(a)



This is an undirected graph.

(b) Adjacency list:

A	→ B → C → D → E
B	→ A → C → F
C	→ A → B → D → G
D	→ A → C → E → G → H
E	→ A → D → H
F	→ B → G
G	→ C → D → F → H
H	→ D → E → G

Adjacency Matrix:

	A	B	C	D	E	F	G	H
A	0	1	1	1	1	0	0	0
B	1	0	1	0	0	1	0	0
C	1	1	0	1	0	0	1	0
D	1	0	1	0	1	0	1	1
E	1	0	0	1	0	0	0	1
F	0	1	0	0	0	0	1	0
G	0	0	1	1	0	1	0	1
H	0	0	0	1	1	0	1	0

(c)

pseudocode for finding mutual friends.

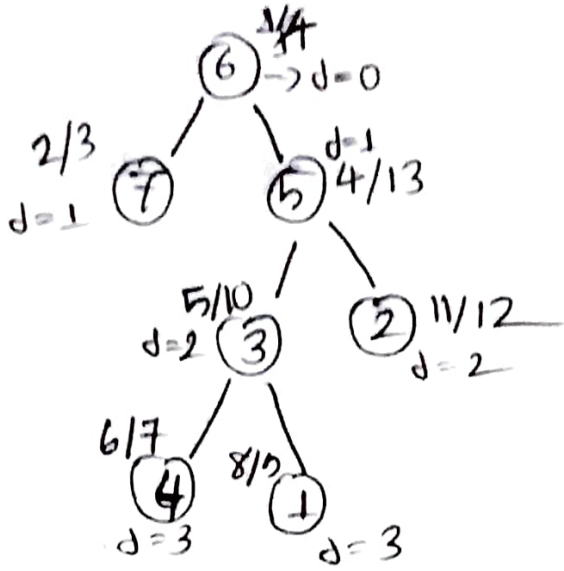
```
def calculate_mutual_friends(graph):
    mutual_friends = { }
    for person1 in graph:
        for person2 in graph:
            if person1 != person2:
                mutuals = len(set(graph[person1]) & set(graph[person2]))
                mutual[(person1, person2)] = mutuals
    return mutual friends
```

Pair	Mutual friends
A, B	{ C }
A, C	{ B, D }
A, D	{ C, E }
A, E	{ D }
B, C	{ A }
B, F	{ G }
G, D	{ A }

Similarly we can find mutual friends for all pairs.

Answer to the question no: 3

(A) DFS Tree:



(B)

Nodes	1	2	3	4	5	6	7
Parent	3	5	5	3	6		6
starting Time	8	11	5	6	4	1	2
Finish time	9	12	10	7	13	14	3
Distance from Root	3	2	2	3	1	0	1

Answer to the question number 4

(A) In an undirected graph:

- degree of the vertex ($deg(v)$) is the number of the edges connected to it.
- If we sum the degrees of all vertices, each node will be counted twice.
- Therefore, $\sum_{v \in V} deg(v) = 2m$.

(A) Answer to the question no: 4

From the graph total degrees

$\deg(A) = 3$

$\deg(B) = 4$

$\deg(C) = 4$

$\deg(D) = 3$

$\deg(E) = 3$

$\deg(F) = 3$

$\deg(G) = 4$

$\deg(H) = 2$

$\deg(S) = 2$

Total = 28 degrees

So, number edges are, $m = 14$

Therefore, $\sum_{v \in V} \deg(v) = 2m = 2 \times 14 = 28$

So, it is applicable.

(B) The maximum number of edges for an undirected graph: [Justified]

$$\text{Max edges} = \frac{n \times (n-1)}{2}$$

we have, 9 vertices, $n = 9$

$$\text{So, max edges} = \frac{9(9-1)}{2} = 36$$

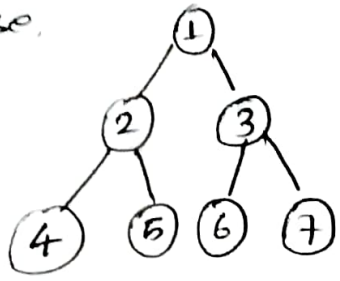
Current edges = 14

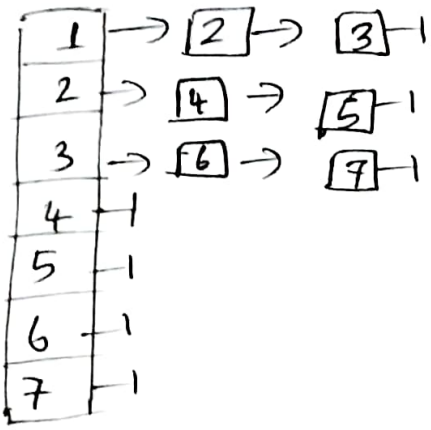
∴ Additional edge = 36 − 14 = 22 (with no loops or multiple edges, keeping the graph simple)

Answer to the question no: 5

(i) Forexample,

Suppose,





Here,
adjacency list

1 : [2, 3]
2 : [4, 5]
3 : [6, 7]
⋮

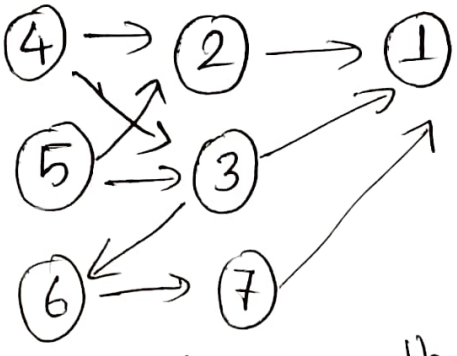
Steps:

1. We can create an empty list on every index and can create a list with the length of nodes given.
2. We can append the child if child found at loop.

(ii)
```
def tree(graph, n):
    size = [0] * (n+1)
    par = [-1] * (n+1)
    def dfs(node, prev):
        for i in data[node]:
            if i != prev:
                par[i] = node
                dfs(i, node)
                size[node] += size[i]
    dfs(1,0)
    return size, par
def query(node, size):
    return size[node]
```

Answer to the question no: 6

Suppose, a directed graph.



From the graph we can see that 1 vertex is reachable from all the other vertices,

Now, for the algorithm steps,

(1) Reverse all the edges directions.
↳ it will take O(E) time complexity

(2) We use DFS on vertex 1 which is reachable from all the other vertices,
↳ it will take O(V+E) time complexity

So, O(V+E) will be the time complexity for this algorithm.

Therefore, for step 1 we reverse path and we see in step 2 if those path can be visited again and it will then give the solution for finding special vertex.