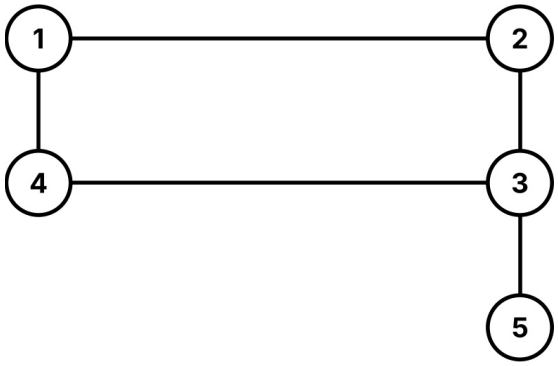
Tutorial on BFS and DFS

[Question1]

Design an algorithm to detect whether an undirected graph contains a cycle using Depth First Search (DFS). The algorithm should run in O(V + E) time, where V is the number of vertices and E is the number of edges.



Hint- Use DFS with a visited[] array and keep track of the parent vertex during traversal. If you encounter an already-visited neighbour that is not the parent, a cycle exists.

Solution:

1. Adjacency List Representation

The adjacency list for the graph is:

Vertex	**Adjacent vertices**
1	2, 4
2	3, 1
3	5, 4, 2
4	3, 1
5	3

2. Key Idea: DFS with a Parent

DFS explores a vertex and then recursively explores each unvisited neighbour. For an undirected graph, simply finding an already-visited neighbour is not enough to prove a cycle: the neighbour may be the vertex from which we just came.

Therefore, the DFS function keeps two parameters: DFS(node, parent). The parent is the previous vertex in the current DFS path.

- Mark the current node as visited.
- For every neighbour of the current node:
- If the neighbour is not visited, recursively call DFS(neighbour, node).
- If the neighbour is already visited and is not the parent, a cycle is found.
- If no such neighbour is found, return false.

3. Pseudocode

```
DFS(node, parent):
   visited[node] = true
   for each neighbor in adjacency_list[node]:
      if visited[neighbor] == false:
         if DFS(neighbor, node) == true:
            return true
      else if neighbor != parent:
         return true      // cycle found
   return false
```

4. Step-by-Step Dry Run

Start DFS from vertex 1. Initially, all vertices are unvisited. The first call is DFS(1, -1), where -1 indicates that vertex 1 has no parent.

Step	DFS call	What happens
Step 1	DFS(1, -1)	Mark 1 as visited. Visit its unvisited neighbour 2.
Step 2	DFS(2, 1)	Mark 2 as visited. Its neighbour 1 is the parent, so ignore that edge. Continue to 3.
Step 3	DFS(3, 2)	Mark 3 as visited. Explore its unvisited neighbour 5 (or 4, depending on neighbour order).
Step 4	DFS(5, 3)	Mark 5 as visited. Its only neighbour is 3, which is its parent. No cycle is reported here; return false.
Step 5	Back to DFS(3, 2)	Continue with neighbour 4. Since 4 is unvisited, call DFS(4, 3).
Step 6	DFS(4, 3)	Mark 4 as visited. Neighbour 3 is the parent. Neighbour 1 is already visited and is not the parent.
Step 7	Cycle detected	Because neighbour 1 is visited and 1 ≠ parent (3), DFS reports a cycle.

5. Time Complexity Analysis

The time complexity of the DFS-based cycle detection algorithm is $O(V + E)$, where:

- V = Number of vertices in the graph.
- E = Number of edges in the graph.

Explanation:

1. Each vertex is visited at most once because the visited[] array prevents repeated visits. Therefore, visiting all vertices takes $O(V)$ time.
2. Each edge is examined at most twice in an undirected graph (once from each endpoint). Therefore, exploring all edges takes $O(2E) = O(E)$ time.

3. Checking whether a neighbour is visited and comparing it with the parent takes $O(1)$ time per adjacency entry.

Hence, the overall time complexity is: O(V+E)

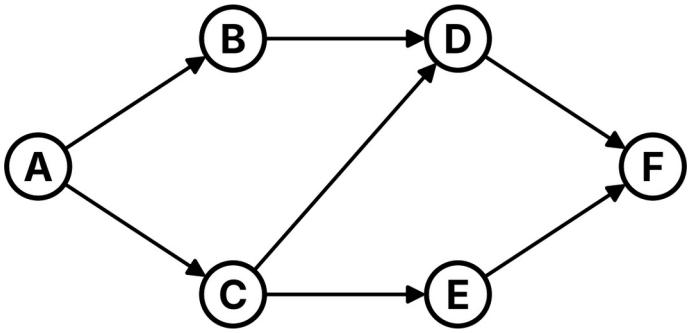
[Question 2]

Given a directed acyclic graph (DAG) G = (V, E), design and implement an algorithm using Depth First Search (DFS) to generate a topological ordering of its vertices.

Consider the following directed graph:

Vertices: A, B, C, D, E, F

Edges: A → B, A → C, B → D, C → D, C → E, D → F, E → F



Perform the following tasks:

i) Apply DFS traversal to the given graph, starting from vertex A.

ii) Maintain and record the discovery time (pre-visit time) and finishing time (post-visit time) of each vertex.

iii) Design an algorithm that generates a topological ordering by arranging vertices in decreasing order of their finishing times.

iv) Display the resulting topological ordering of the graph.

v) Analyse the time and space complexity of the proposed algorithm.

Hint- Use DFS and record the finishing time (post-visit time) of each vertex. Push each vertex onto a stack after exploring all its neighbours; popping the stack gives the topological ordering.

Solution-

Step 1 – Adjacency list

Vertex	Neighbours (alphabetical)
A	B, C
B	D
C	D, E
D	F
E	F
F	—

Step 2 – DFS trace from A (timer starts at t = 0)

- Visit A, pre(A) = 1 → go to B, pre(B) = 2 → go to D, pre(D) = 3 → go to F, pre(F) = 4.
- F has no outgoing edges: post(F) = 5. Back in D: post(D) = 6. Back in B: post(B) = 7.
- Back in A, next neighbour C: pre(C) = 8. C→D is already visited. Go to E, pre(E) = 9.
- E→F is already visited: post(E) = 10. Back in C: post(C) = 11. Back in A: post(A) = 12.

- All vertices are visited, so no new DFS start is needed.

Pre / Post table

Vertex	Pre (discovery)	Post (finish)
A	1	12
B	2	7
C	8	11
D	3	6
E	9	10
F	4	5

Step 3 – Algorithm

```
TopologicalSort(G):
    time = 0; order = empty stack
    for each vertex v in V (alphabetical):
        if v is not visited: DFS(v)
    return order (read from top of stack)

DFS(u):
    visited[u] = true; pre[u] = ++time
    for each v in Adj[u] (alphabetical):
        if v is not visited: DFS(v)
    post[u] = ++time
    push u onto order        // vertices pushed in increasing post time
```

Step 4 – Topological ordering

Arrange vertices by decreasing post (finishing) time: A (12), C (11), E (10), B (7), D (6), F (5).

Topological order: A → C → E → B → D → F

Edge check

Edge u→v	post(u)	post(v)	post(u) > post(v)
A→B	12	7	✓
A→C	12	11	✓
B→D	7	6	✓
C→D	11	6	✓
C→E	11	10	✓
D→F	6	5	✓
E→F	10	5	✓

Step 5 – Why the ordering is valid

Claim: for every edge (u → v) in a DAG, post(u) > post(v). Consider the moment DFS first examines the edge u → v. There are only three cases:

- v is unvisited: DFS(v) starts and must finish before control returns to u, so post(v) < post(u).
- v is already finished: post(v) was assigned earlier than post(u), so post(v) < post(u).
- v is still in progress (an ancestor of u): this is a back edge and would form a cycle. A DAG has no cycle, so this case cannot occur.

Therefore sorting by decreasing post time puts u before v for every edge, which is exactly the topological-ordering property.

Step 6 – Complexity

	Cost	Reason
Time	O(V + E)	Each vertex is visited once and each edge is examined once; ordering is produced as vertices finish, so no extra sort is needed.

	Cost	Reason
Space	O(V + E)	Adjacency list takes O(V + E); visited, pre, post and the output stack take O(V); recursion depth is at most O(V).

For this graph: V = 6, E = 7, so about 13 basic steps.

[Question 3]

Given an undirected graph $G = (V, E)$, design and implement an algorithm using Depth First Search (DFS) to count the total number of connected components in the graph.

Consider the following undirected graph:

Vertices: 1, 2, 3, 4, 5, 6, 7, 8

Edges:

- 1 – 2
- 2 – 3
- 3 – 1
- 4 – 5
- 5 – 6
- 7 – 8

Perform the following tasks:

i) Represent the given graph using an adjacency-list representation.

ii) Apply DFS traversal to identify all connected components of the graph.

iii) Design an algorithm that counts the number of connected components using DFS.

iv) Display the vertices belonging to each connected component and the total number of connected components.

v) Analyse the time and space complexity of the proposed algorithm.

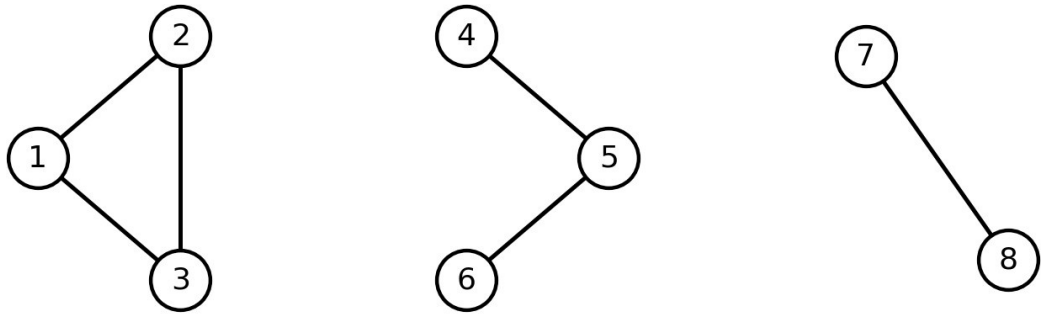
Hint: Use a visited[] array and iterate through all vertices. Whenever an unvisited vertex is found, increment the component counter and start a new DFS traversal from that vertex.

Solution-

Graph Structure

Vertices: 1, 2, 3, 4, 5, 6, 7, 8

Edges: (1–2), (2–3), (3–1), (4–5), (5–6), (7–8)



Component 1 Component 2 Component 3

Step 1 – Adjacency list

Vertex	Adjacent vertices
1	2, 3
2	1, 3
3	1, 2

Vertex	Adjacent vertices
4	5
5	4, 6
6	5
7	8
8	7

Step 2 – Key Idea: DFS with Component Counter

The algorithm relies on Depth First Search to discover all vertices within a single connected component. A visited[] array tracks nodes that have already been explored.

- Mark all nodes as unvisited and initialize the component counter to zero.
- Iterate through every vertex in the graph.
- Upon finding an unvisited node, increment the component counter and trigger a new DFS traversal from that vertex.
- DFS explores and marks all vertices reachable from the root vertex.
- Repeat until every vertex is processed; each fresh DFS execution reveals one distinct component.

Step 3 – Pseudocode

```
COUNT_COMPONENTS(G):
    for each vertex v in V:
        visited[v] = false

    count = 0

    for each vertex v in V:
        if visited[v] == false:
            count = count + 1
            DFS(v)
```

```
    return count

DFS(u):
    visited[u] = true

    for each neighbour v in Adj[u]:
        if visited[v] == false:
            DFS(v)
```

Step 4 – Step-by-Step Dry Run

Initially, all vertices remain unvisited. Process vertices in ascending numerical order.

DFS call	What happens	Component	Count
DFS(1)	Visit 1, then explore neighbour 2, then 3. All three vertices form a connected sub-graph and are marked visited.	{1, 2, 3}	1
DFS(4)	Vertex 4 is unvisited. Start a new DFS: visit 4, then 5, and then 6.	{4, 5, 6}	2
DFS(7)	Vertex 7 is unvisited. Start a new DFS call and traverse to neighbour 8.	{7, 8}	3

Following the DFS initiated at vertex 7, all 8 vertices have been visited. Thus, no further searches are triggered.

Step 5 – Component Output

- Component 1: {1, 2, 3}
- Component 2: {4, 5, 6}
- Component 3: {7, 8}

Total connected components: 3

Step 6 – Complexity

Let V represent the vertex count and E represent the edge count.

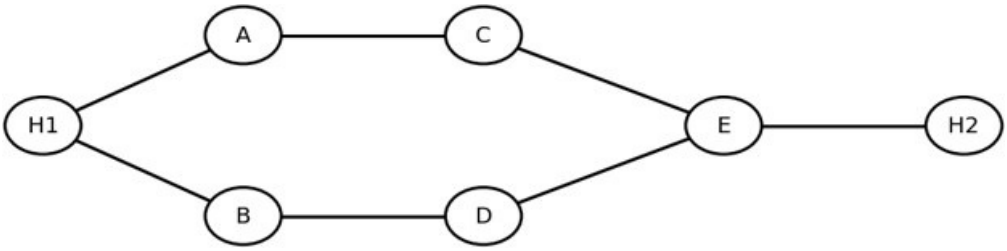
Operation	Cost	Reason
Array Initialization	O(V)	Initialising visited status takes O(1) time per vertex.
DFS Traversal	O(V + E)	Each node is visited once and each edge is traversed twice in an undirected graph.
Outer Loop	O(V)	Iterating through all vertices requires linear time.

Hence, the overall time complexity is: O(V + E)

Auxiliary space requires O(V) for recursion depth and visited flags. Storing the graph as an adjacency list takes O(V + E), yielding an overall space complexity of O(V + E).

[Question 4]

A city is represented by an unweighted, undirected graph G = (V, E), where vertices are intersections and edges are roads. A set H ⊆ V contains emergency facilities. Design an algorithm that determines, for every intersection v, the minimum number of roads needed to reach the nearest facility. If two facilities are equally near, either one may be reported.



Illustrative road network; H1 and H2 are emergency facilities.

Solution:

A usual BFS starts from one source. Here, there are several possible sources: all emergency facilities. Put every facility into the queue before the search begins and assign distance 0 to each one. This makes the search expand outwards from all facilities simultaneously.

- Initialise dist[v] = ∞ and owner[v] = NIL for every vertex v.
- For each facility h ∈ H, set dist[h] = 0, owner[h] = h, and enqueue h.
- When an unvisited neighbour v is first reached from u, set dist[v] = dist[u] + 1 and owner[v] = owner[u].
- The owner value records which facility first reached v. A tie may be resolved by whichever facility is processed first.

Pseudocode

```
NEAREST_FACILITY(G, H):
    for each vertex v in V:
    dist[v] = infinity
    owner[v] = NIL

    Q = empty queue
    for each facility h in H:
    dist[h] = 0
    owner[h] = h
```

```
enqueue(Q, h)

while Q is not empty:
u = dequeue(Q)
for each neighbour v of u:
        if dist[v] == infinity:
        dist[v] = dist[u] + 1
        owner[v] = owner[u]
        enqueue(Q, v)

return dist, owner
```

Worked BFS execution

Consider the displayed network. The facilities H1 and H2 are both inserted initially. Assume the adjacency lists are examined in the order shown by the diagram.

Step	**Vertex removed from queue**	**Newly discovered vertices**	**Queue after processing**
Initial	—	H1, H2 (both distance 0)	[H1, H2]
1	H1	A, B (distance 1 from H1)	[H2, A, B]
2	H2	E (distance 1 from H2)	[A, B, E]
3	A	C (distance 2 from H1)	[B, E, C]
4	B	D (distance 2 from H1)	[E, C, D]
5	E	No new vertex; C and D are already discovered	[C, D]
6–7	C, then D	No new vertices	[]

Final distances: dist(H1) = dist(H2) = 0; dist(A) = dist(B) = dist(E) = 1; dist(C) = dist(D) = 2.

Thus A and B are assigned to H1, E is assigned to H2, and C and D are assigned to H1 under this queue order. If a vertex is equally close to two facilities, the assignment may depend on the order in which the facilities enter the queue.

[Question 5]

You are given a start word, a target word, and a dictionary containing words of the same length. In one move, you may change exactly one letter. After each move, the resulting word must be in the dictionary. The start word need not be in the dictionary. Design an algorithm to find a valid sequence from the start word to the target word using the fewest moves. If no such sequence exists, report that it is impossible.

Example: Start = HIT, target = DIG, dictionary = {HOT, DOT, DOG, DIG}. A valid sequence is HIT → HOT → DOT → DOG → DIG. This requires four moves.



A word graph: each edge joins words that differ in exactly one character.

Solution:

Represent every dictionary word as a vertex. Connect two words if they differ in exactly one character. Each edge represents one legal move, so a valid transformation is a path in this graph. Since every move has equal cost, BFS finds a path with the fewest moves. Store parent[word] when a word is first discovered so that the sequence can be reconstructed.

To avoid comparing every pair of words, use wildcard-pattern buckets. For example, HOT produces *OT, H*T, and HO*. Words in the same bucket differ at only the position replaced by the wildcard, so they are neighbours. Process each bucket only once.

Pseudocode

```
WORD_LADDER(start, target, dictionary):
    if target is not in dictionary:
    return "No sequence"

    W = dictionary union {start}
    build buckets: for each word w and each position i,
    replace character i by * and add w to that pattern's bucket

    mark every word unvisited
    parent[w] = NIL for every word w
    mark start visited
    enqueue(start)

    while Q is not empty:
    w = dequeue(Q)
    if w == target:
            break

    for each wildcard pattern p of w:
            if p has not been processed:
            mark p processed
            for each word z in bucket[p]:
                    if z is unvisited:
                    mark z visited
                    parent[z] = w
                    enqueue(z)

    if target is unvisited:
```

```
    return "No sequence"
    return the sequence obtained by following parent[] from target to start,
            then reversing it
```

Worked BFS execution

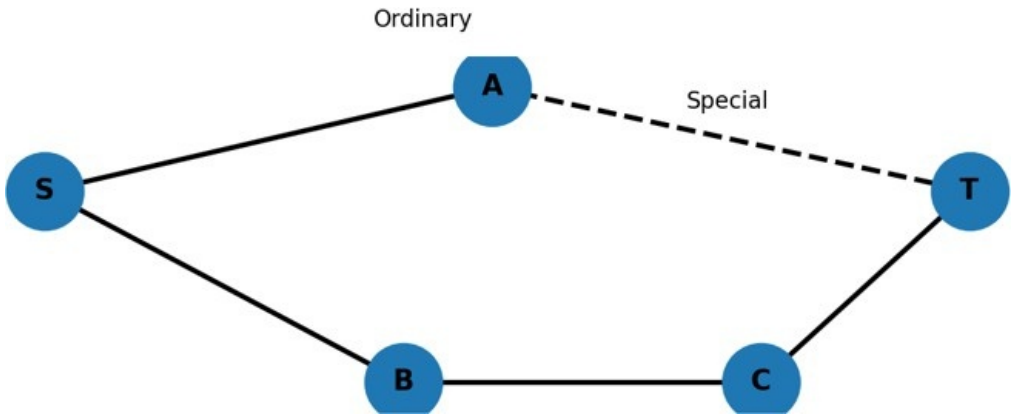
For the given example, the only available one-letter transitions form the chain HIT — HOT — DOT — DOG — DIG.

Step	Word removed from queue	Newly discovered word	Queue after processing
Initial	—	HIT is the start	[HIT]
1	HIT	HOT; parent[HOT] = HIT	[HOT]
2	HOT	DOT; parent[DOT] = HOT	[DOT]
3	DOT	DOG; parent[DOG] = DOT	[DOG]
4	DOG	DIG; parent[DIG] = DOG	[DIG]
5	DIG	Target reached	[]

Following parent pointers backwards gives DIG ← DOG ← DOT ← HOT ← HIT. Reversing this sequence gives HIT → HOT → DOT → DOG → DIG, which uses four moves.

[Question 6]

A transport network is an unweighted, undirected graph. Each road is either ordinary or special. A vehicle may use any number of ordinary roads but at most one special road. Design an algorithm to find a route from source S to destination T that uses the minimum number of roads while respecting this restriction. Return the route, or report that no valid route exists.



Solid edges are ordinary roads; the dashed edge A–T is special.

In this example, S → A → T is a valid route of two roads: it uses one ordinary road and one special road. The alternative S → B → C → T uses three ordinary roads.

Solution:

A vertex alone is not enough to describe the search position. When the vehicle reaches the same intersection, the available future moves depend on whether it has already used the special road. Therefore, create two states for each original vertex v:

- (v, 0): the vehicle is at v and has not used the special road.
- (v, 1): the vehicle is at v and has already used the special road.

For an ordinary road u–v, add transitions (u,0) → (v,0) and (u,1) → (v,1). For a special road u–v, add transitions (u,0) → (v,1) and (v,0) → (u,1), if the road is undirected. There is no special-road transition from a state whose second component is 1.

Run BFS from (S,0). The destination can be reached in either state: (T,0) means no special road was used, while (T,1) means exactly one was used. Choose whichever has the smaller finite distance.

Pseudocode

```
SHORTEST_WITH_ONE_SPECIAL(G, S, T):
    for each vertex v and b in {0, 1}:
    dist[v,b] = infinity
    parent[v,b] = NIL

    dist[S,0] = 0
    enqueue(Q, (S,0))

    while Q is not empty:
    (u,b) = dequeue(Q)

    for each ordinary neighbour v of u:
            if dist[v,b] == infinity:
            dist[v,b] = dist[u,b] + 1
            parent[v,b] = (u,b)
            enqueue(Q, (v,b))

    if b == 0:
            for each special neighbour v of u:
            if dist[v,1] == infinity:
                    dist[v,1] = dist[u,0] + 1
                    parent[v,1] = (u,0)
                    enqueue(Q, (v,1))

    choose (T,b) with the smaller finite dist[T,b]
    if both distances are infinity:
    return "No valid route"
    return reconstruct route using parent states
```

Worked BFS execution

For the network shown, BFS starts at (S,0). The state changes only when a special road is used. The following trace shows the shortest route to the destination.

State removed	Road taken	New state	Distance
(S,0)	Ordinary S–A	(A,0)	1
(S,0)	Ordinary S–B	(B,0)	1
(A,0)	Special A–T	(T,1)	2
(B,0)	Ordinary B–C	(C,0)	2
(T,1)	Destination reached	—	2
(C,0)	Ordinary C–T	(T,0)	3

The two destination states have distances dist[T,1] = 2 and dist[T,0] = 3. The algorithm therefore returns S → A → T, which is the shortest valid route.