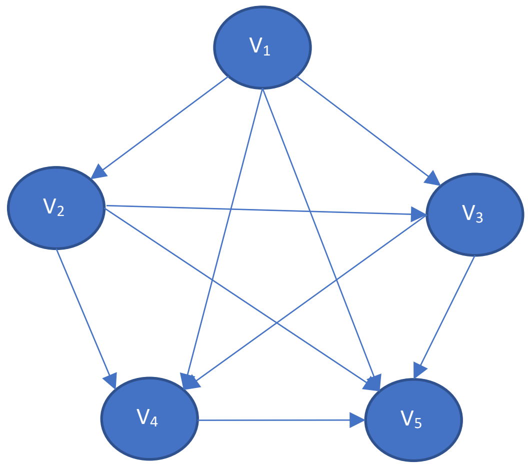
Answer to Question 1



Graph G has five vertices: v_1, v_2, v_3, v_4, and v_5. Adjacency list can be used in the linked representation to store graph G.

Algorithm to store the graph in the form of a Linked list in the memory:

- Adjacency graph representation of a graph $G = (V, E)$, where V is the number of vertices and E is the number of edges, consists of an array Adj of |V| lists.
- Adj consists of one list per vertex.
- The adjacency list, for each $u \in V$, $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$. This means, $Adj[u]$ consists of all the vertices adjacent to u in G.

Pseudocode:

#Initialize an empty

dictionary graph = {};

```
Add_edge(u, v):
        if u in graph { graph[u].append(v); } #add v to list of neighbors for u
        else { graph[u] = [v]; } #create a new list with v as the first neighbor
```

Add_edge(v_1, v_2);

Add_edge(v_1, v_3);

Add_edge(v_1, v_4);

Add_edge(v_1, v_5);

Add_edge(v_2, v_3);

Add_edge(v_2, v_4);

Add_edge(v_2, v_5);

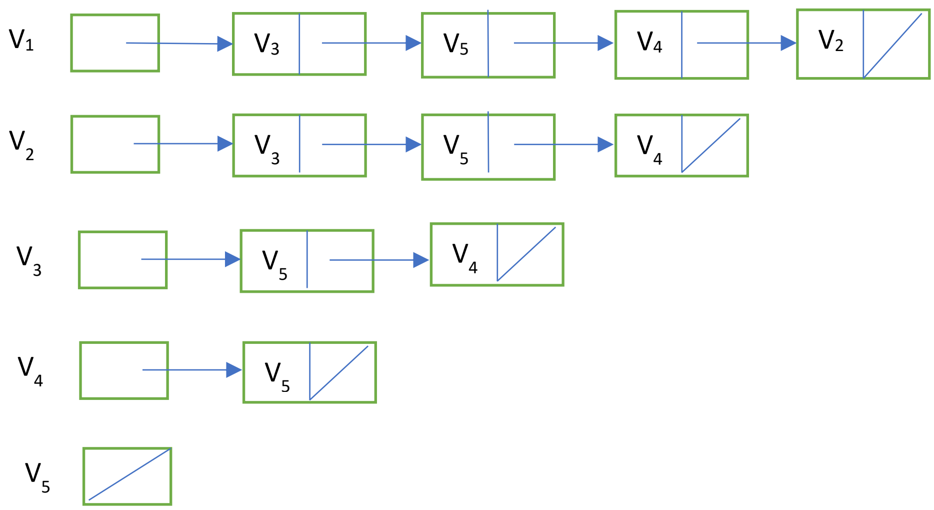
Add_edge(v_3, v_4);

Add_edge(v_3, v_5);

Add_edge(v_4, v_5);

Adjacency lists may also be implemented with pointers to the vertices. The adjacency lists represent the edges of a graph. In the directed graph above, G, the sum of the lengths of all the adjacency lists is |E|, since an edge of the form (u, v) is represented by having v appear in Adj[u]. The space complexity of the above algorithm is $\theta(V + E)$.

Adjacency List Representation:



Answer to Question 2

The degree of a vertex in a directed graph is its in-degree plus its out-degree. In-degree is the number of incoming edges and out-degree is the number of outgoing edges of a vertex in a graph. Queue data structure is used that follows first-in-first-out method.

Algorithm idea:

Step 1: Compute and initialize the in-degree for each vertex present in the Directed Acyclic Graph with number of visited nodes = 0.

Step 2: Find all the vertices with in-degree = 0 and add them into a queue (perform enqueue operation).

Step 3: Remove a vertex from the queue (perform dequeue operation) and then perform the following –

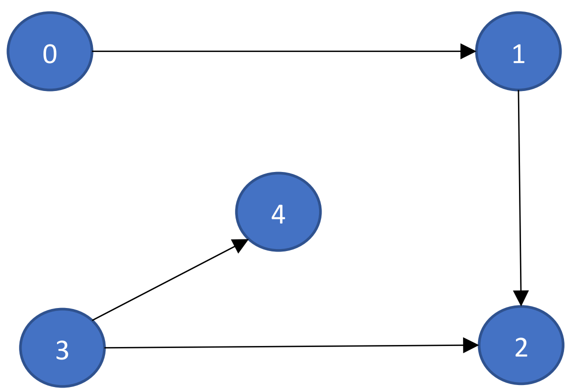
1. Increment the count of visited nodes by 1.

2. For all the neighboring nodes, decrease in-degree by 1.
3. Check if in-degree of neighboring nodes are reduced to zero, then add it to the queue (perform enqueue operation).

Step 4: Repeat step 3 until the queue is empty.

The count of the visited nodes must be equal to the number of nodes in the directed acyclic graph.

Example :



Output: 0 3 4 1 2

Analysis:

Since the graph is acyclic, there must exist a vertex with in-degree = 0 and a vertex with out-degree = 0.

We need a list for each vertex (in-degree). The construction of this linked list can be done in $O(|V| + |E|)$ time. This means, Step 1, calculation of each in-degree can be done in $O(|V| + |E|)$.The enqueue and dequeue operations can be done in constant time $O(1)$ and we have to do this for each vertex at least once. So, total time for queueing is $O(V)$. The insertion and deletion of the doubly linked list can be done at $O(1)$. We must perform this for each child of each vertex at least one, so it has to be done |E| times. At each step we are outputting an element with in-degree = 0, with respect to all the vertices that had not been finished.

Therefore, the total runtime is $O(|V| + |E|)$.

Answer to Question 3

Algorithm:

Step 1: Explore G and generate an interval I_v: $[d(v), f(v)]$ for each node v.

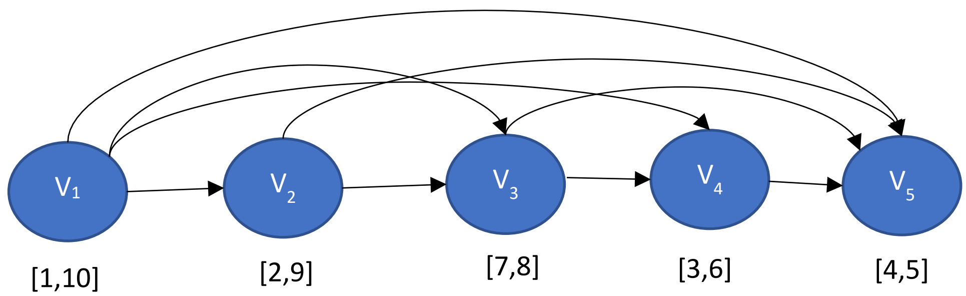
Step 2: Generate a topological order of G.

Step 3: Generate an interval sequence for each node v along the reverse of the topological order as follows:

Let $v_1, v_2, \dots, v_k$ be the children of v.

Let S_i be the interval sequence generated for $v_i (i = 1, \dots, k)$.

Merge $S_1, S_2, \dots, S_k$ and I_v to generate S_v.



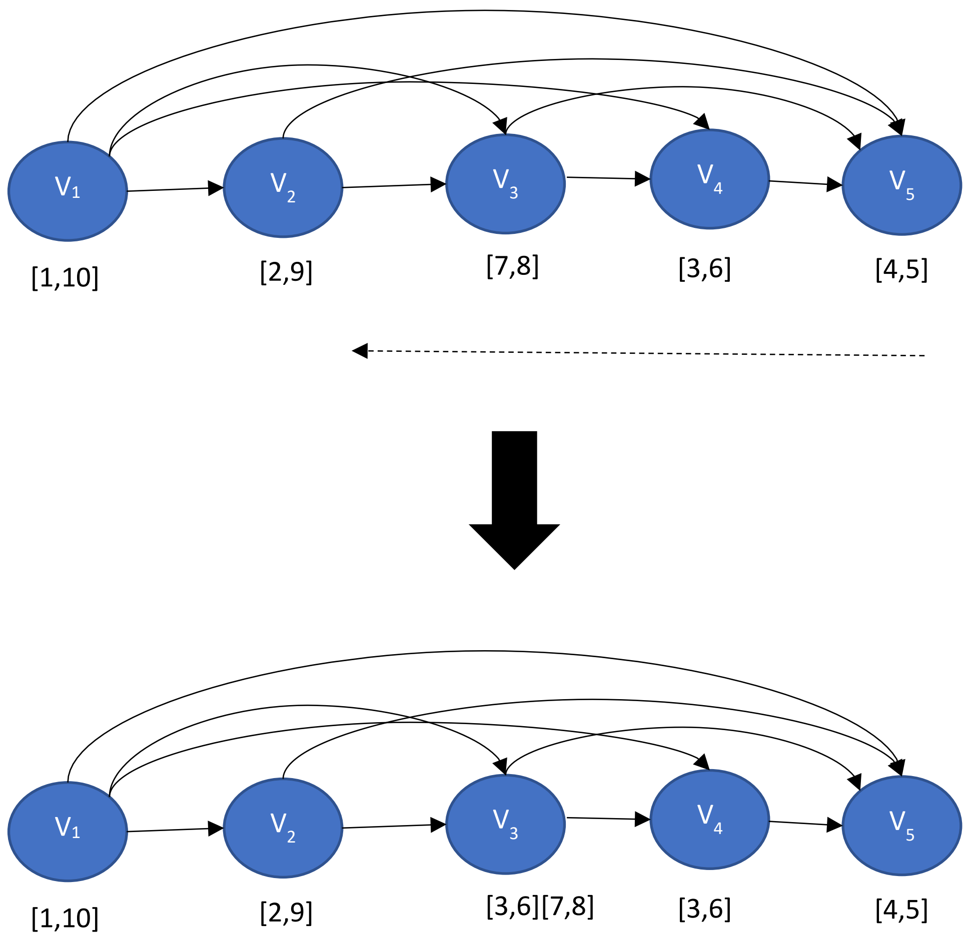
Merge Operation of two interval sequences S_1 and S_2:-

Let $S_1 = [a_1, b_1][a_2, b_2] \ldots [a_l, b_l]$ and $S_2 = [c_1, d_1][c_2, d_2] \ldots [c_k, d_k]$

Merge S_1 and S_2 to generate:

$$S = [x_1, y_1][x_2, y_2] \ldots [x_m, y_m] \text{ (topologically sorted)}$$

Here, each $[x_p, y_p]$ is some $[a_i, b_i]$ or some $[c_j, d_j]$ but each $[x_p, y_p]$ is not a subinterval of any other $[x_q, y_q]$ and vice versa.



Merging S_1 and S_2 would store the result in S_1.

Merge(S_1, S_2):

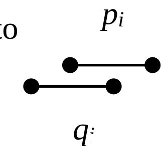
$S_1 = p_1 p_2 \dots p_l = [a_1, b_1][a_2, b_2] \dots [a_l, b_l]$

$S_2 = q_1 q_2 \dots q_m = [c_1, d_1][c_2, d_2] \dots [c_k, d_k]$ #Assuming that both of them are sorted in increasing order according to the starting time points

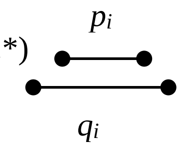
Initially, $i = 1 \; and \; j = 1$

#There are five possible cases:

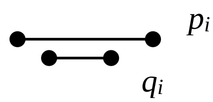
Case 1: If $p_i.a > q_j.c$ and $p_i.b > q_j.d$: insert q_j into S_1 after p_{i-1} and before p_i and move to q_{j+1}



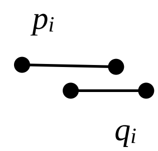
Case 2: If $p_i.a > q_j.c$ and $p_i.b < q_j.d$: remove p_i from S_1 and move to p_{i+1}. (*p_i is covered by q_j.*)



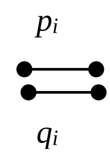
Case 3: If $p_i.a < q_j.c$ and $p_i.b > q_j.d$, ignore q_j and move to q_{j+1}. (*q_j is covered by p_i; but it should not be removed from S_2.*)



Case 4: If $p_i.a < q_j.c$ and $p_i.b < q_j.d$, ignore p_i and move to p_{i+1}.



Case 5: If $p_i.a = q_j.c$ and $p_i.b = q_j.d$ ignore both p_i and q_j, and move to p_{i+1} and q_{j+1}, respectively.



Example:

p

S_1 : [7,7][8,9]

S_2 : [3,4][8,5][10,11

q

p

S_1 : [3,4][7,7][8,9]

S_2 : [3,4][5,8][10,11]

q

p

S_1 : [3,4][7,7][8,9]

S_2 : [3,4][5,8][10,11]

q

p

S_1 : [3,4][7,7][5,8][8,9]

S_2 : [3,4][5,8][10,11]

q

p

S_1 : [3,4][7,7][5,8][8,9][10,11]

S_2 : [3,4][5,8][10,11]

q