### Algorithm: Tarjan's Algorithm

Princess Elara needed to find all the critical points in her underwater city, Aquapolis, to protect it from sea monster attacks. She used Tarjan's Algorithm to find strongly connected components in the city's infrastructure.

#### Initialize Data Structures:

* Princess Elara used a trident (stack) to keep track of nodes.
* She also used a magical compass (array) to keep track of discovery and low values.

#### Explore the City:

* She explored each part of the city, marking nodes and discovering strongly connected components.

#### Implementation:

| **def** tarjans\_scc(city: Dict[int, List[int]]) -> List[List[int]]:  **def** dfs(v):  **nonlocal** index  discovery[v] = low[v] = index  index += 1  stack.append(v)  on\_stack[v] = **True**  **for** w **in** city[v]:  **if** discovery[w] == -1:  dfs(w)  low[v] = min(low[v], low[w])  **elif** on\_stack[w]:  low[v] = min(low[v], discovery[w])  **if** low[v] == discovery[v]:  component = []  **while** **True**:  w = stack.pop()  on\_stack[w] = **False**  component.append(w)  **if** w == v:  **break**  scc.append(component)  discovery = [-1] \* len(city)  low = [-1] \* len(city)  on\_stack = [**False**] \* len(city)  stack = []  scc = []  index = 0  **for** v **in** range(len(city)):  **if** discovery[v] == -1:  dfs(v)  **return** scc  *# Example usage:*  city = {  0: [1],  1: [2],  2: [0, 3],  3: [4],  4: [5],  5: [3]  }  print(tarjans\_scc(city)) *# Output: Strongly connected components* |
| --- |

#### Explanation:

Initialize:

* discovery, low: Arrays to track discovery and low values.
* stack, on\_stack: A stack and boolean array to track nodes on the stack.

Explore the City:

* Princess Elara explored each part of the city, marking nodes and discovering strongly connected components.### Algorithm: Topological Sort

Topological sorting of a directed graph is a linear ordering of its vertices such that for every directed edge (uv), vertex (u) comes before (v) in the ordering.

#### Initialize Data Structures:

* Use a stack to store the topological order.
* Use a set to keep track of visited vertices.

#### Depth-First Search:

* Perform a depth-first search on each unvisited vertex, pushing vertices onto the stack after all their neighbors have been visited.

#### Retrieve the Result:

* The stack contains the topological order in reverse.

#### Implementation:

| **def** topological\_sort(graph: Dict[int, List[int]]) -> List[int]:  **def** dfs(node):  visited.add(node)  **for** neighbor **in** graph[node]:  **if** neighbor **not** **in** visited:  dfs(neighbor)  stack.append(node)  visited = set()  stack = []  **for** node **in** graph:  **if** node **not** **in** visited:  dfs(node)  **return** stack[::-1]  *# Example usage:*  graph = {  0: [1, 2],  1: [3],  2: [3],  3: []  }  print(topological\_sort(graph)) *# Output: Topological order* |
| --- |

#### Explanation:

Initialize:

* stack: A stack to store the topological order.
* visited: A set to keep track of visited vertices.

Depth-First Search:

* Perform a depth-first search on each unvisited vertex, pushing vertices onto the stack after all their neighbors have been visited.

Retrieve the Result:

### The stack contains the topological order in reverse.