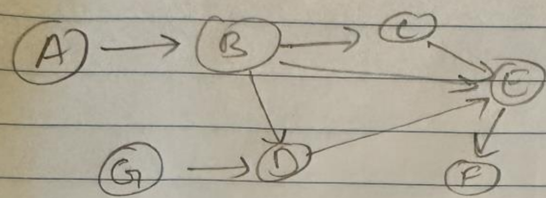
Q1.

Q1.



B

F

	inlist	outlist
A	[ ]	[B]
B	[A]	[C, D, E]
C	[B]	[E]
D	[B, G]	[E]
E	[C, B, D]	[F]
F	[E]	[ ]
G	[ ]	[D]

stack : (A, ~~G~~, ~~D~~, ~~E~~, ~~F~~), (~~A~~, ~~B~~, ~~C~~) → Stack

visited : (G, D, E, F, A, B, C) → Set

cycle : (~~G~~, ~~D~~, ~~E~~, ~~F~~), (~~A~~, ~~B~~, ~~C~~) → set

result : (F, E, D, G, C, B, A) → Stack

Topological Sort => A, B, C, G, D, E F

Q2.

```
Algorithm: IsReachableFrom(G, u, v)
  Input: A directed graph G, vertices u, v in G
  Output: TRUE if there is a directed path from u to v in G, false otherwise.

  inlist <- hashMap()
  outlist <- hashMap()

  for (v1,v2) in G do
    inlist.getOrDefault(v1, new list()).add(v2)
    outlist.getOrDefault(v2, new list()).add(v1)

  startinglist <- new Stack()
  startinglist.push(u)
  visited <- new set()

  while !startinglist.isEmpty() do
    p <- startinglist.peek()
    if(p==v) return true;
    if p in visited then
      startinglist.pop()
    else
      visited.add(p)
      for item in outlist.get(p) then
        if item not in visited then
          startinglist.push(item)
  return false;
```

Time Complexity: O(m + n)
- for inlist and outlist generation O(m) ; m is number of edges
- for while loop at worst case we visit all the vertices once O(n); n is the number of vertices

Space complexity: O(n) + O(n) + O(m) + O(m) -> O(m + n)
- at worst case set will contain all vertices -> O(n) ; n is the number of vertices
- at worst case stack will contain all vertices -> O(n) ; n is the number of vertices
- inlist and outlist -> O(m) + O (m)

Q3.

```
private static boolean dfs(int node,List<List<Integer>> adjList, Set<Integer> visited,
Set<Integer> path,List<Integer> result){
    if(path.contains(node)) return false; // cycle detected

    if(visited.contains(node)) return true; // no need to visit anymore

    List<Integer> dependencies = adjList.get(node);

    path.add(node);

    for(int dep: dependencies){
        if(!dfs(dep,adjList,visited,path,result)){
            return false;
        }
    }

    path.remove(node);
    visited.add(node);
    result.add(node);
    return true;
}

public static int[] findOrder(int numCourses, int[][] prerequisites) {
    List<List<Integer>> adjList = new ArrayList<>();

    // initialize in list and out list/adjacency list
    for(int i=0;i<numCourses;i++){
        adjList.add(new ArrayList<>());
    }

    // adjacency list
    for(int[] pre: prerequisites){
        adjList.get(pre[0]).add(pre[1]);
```

```
    }

    Set<Integer> visited = new HashSet<>();
    Set<Integer> currentPath = new HashSet<>();

    List<Integer> result = new ArrayList<>(numCourses);

    for(int i=0;i<numCourses;i++){
      if(!dfs(i,adjList,visited,currentPath,result)){
        return new int[]{}; // cycle detected not possible to take all courses
      };
    }


    if(visited.size() != numCourses){
      // cannot take all courses
      return new int[]{};
    }


    return result.stream().mapToInt(Integer::intValue).toArray();
}
```

Q4.

```
class Solution {
public boolean dfs(int node,List<List<Integer>> adjList, Set<Integer> visited,
Set<Integer> cycle){
if(cycle.contains(node)) return false;

if(visited.contains(node)) return true;

cycle.add(node);
for(int n: adjList.get(node)){
if(!dfs(n,adjList,visited,cycle)){
return false;
}
}

cycle.remove(node);
```

```
visited.add(node);
return true;
}

public boolean canFinish(int numCourses, int[][] prerequisites) {
List<List<Integer>> adjList = new ArrayList();

for(int i=0;i<numCourses;i++){
adjList.add(i,new ArrayList());
}

for(int[] pre: prerequisites){
adjList.get(pre[0]).add(pre[1]);
}

Set<Integer> visited = new HashSet();
Set<Integer> cycle = new HashSet();
for(int i=0;i<numCourses;i++){
if(!dfs(i,adjList,visited,cycle)){
return false; // cycle detected
}
}

return true;
}
}
```