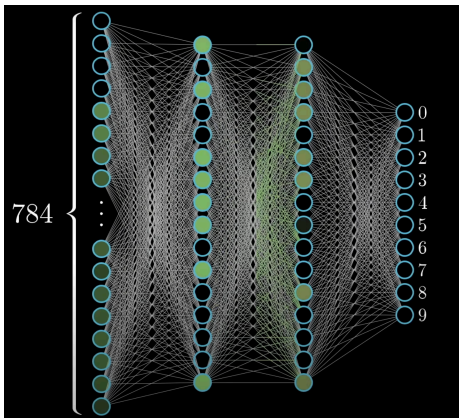
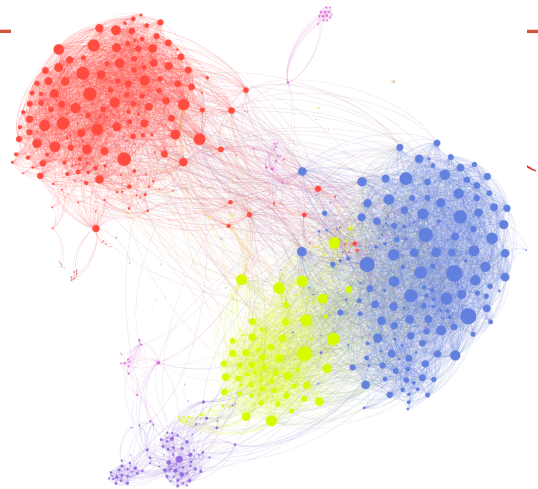
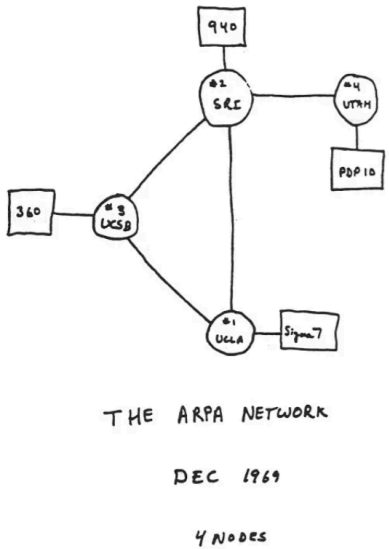
# GRAPH SEARCH – DEPTH FIRST

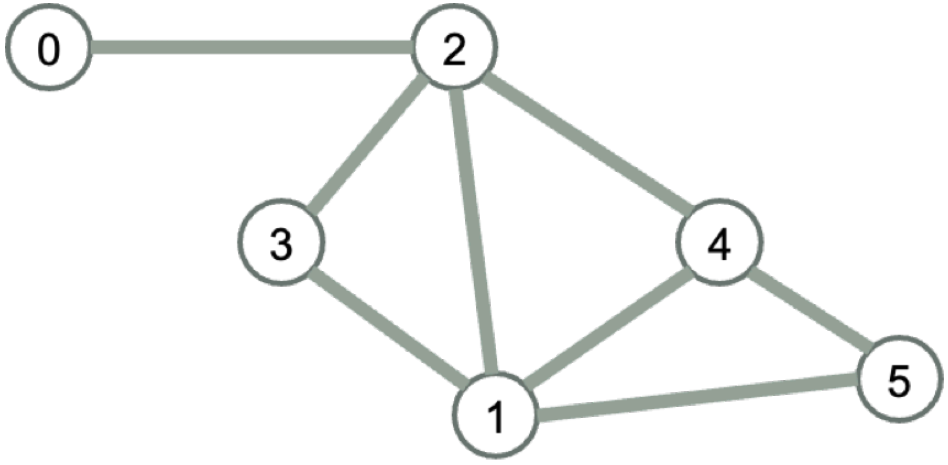




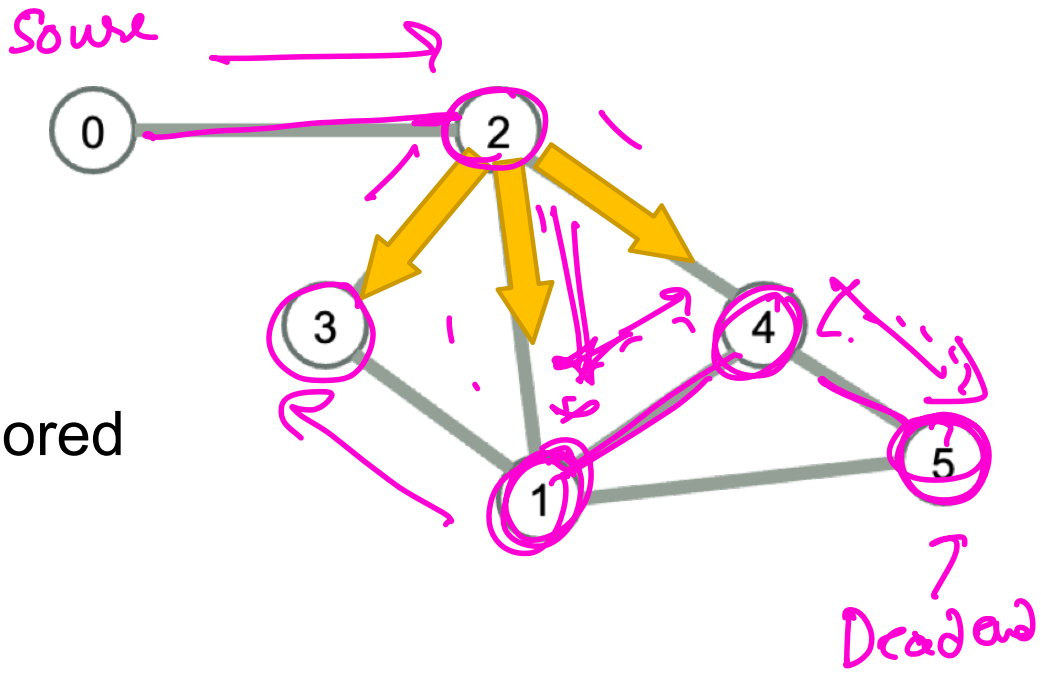
# Graph search: general approach

```
Keep track of all areas discovered

While there is an unexplored path, follow path
```



# Systematize the Search with DFS

**Depth-First Search** explores a graph by following one branch as far as it can go before backtracking. It uses a stack (explicit or via recursion) to remember where to return.



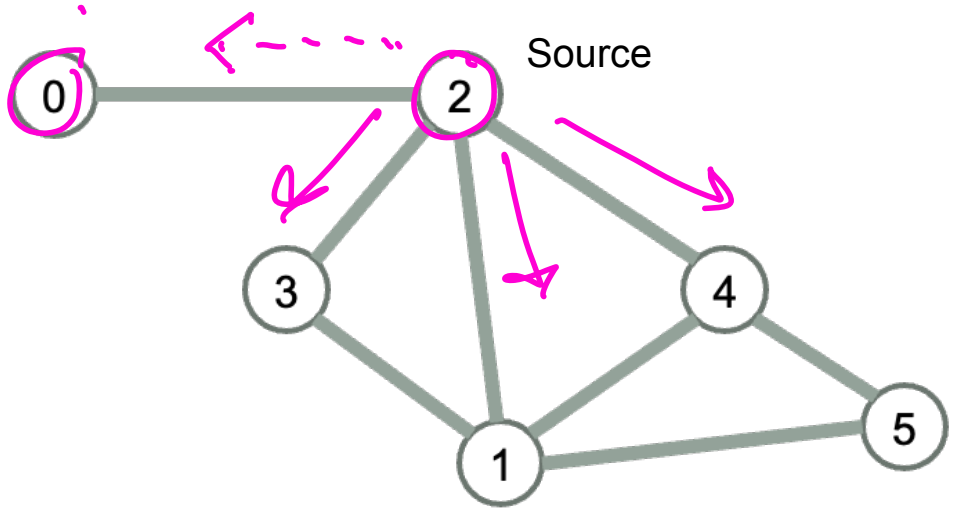

Need to keep track of:

– Which vertices discovered

– Which edges have yet to be explored

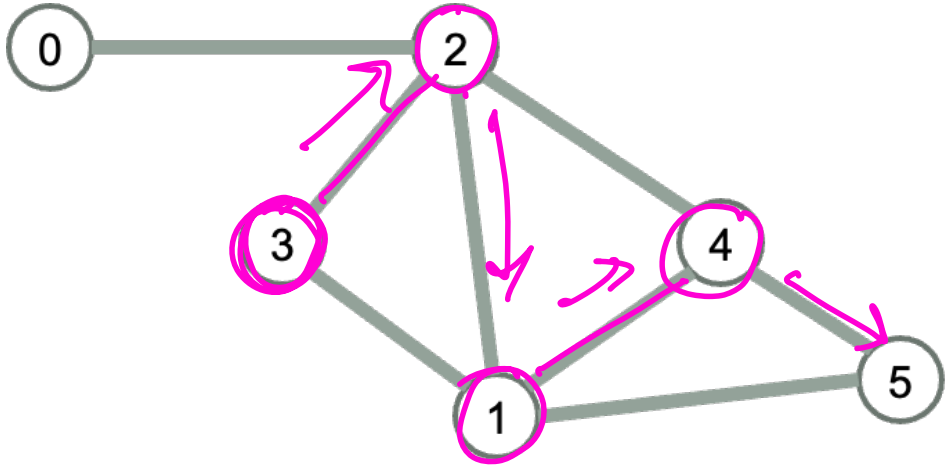
# Explore – Depth First

```
exploreDFS(v)

  v.visited ← true

  For each edge (v,w)

    If not w.visited

      exploreDFS(w)
```



# Explore (Depth First)

Search as far down a single path as possible, backtrack as needed




**Predict the visit order for exploreDFS(3). Which is correct?**

**A) 3, 2, 1, 0, 4, 5**
**B) 3, 1, 4, 2, 0, 5**
**C) 3, 1, 2, 0, 4, 5**
**D) 3, 2, 4, 5, 1, 0**
**E) All except A**

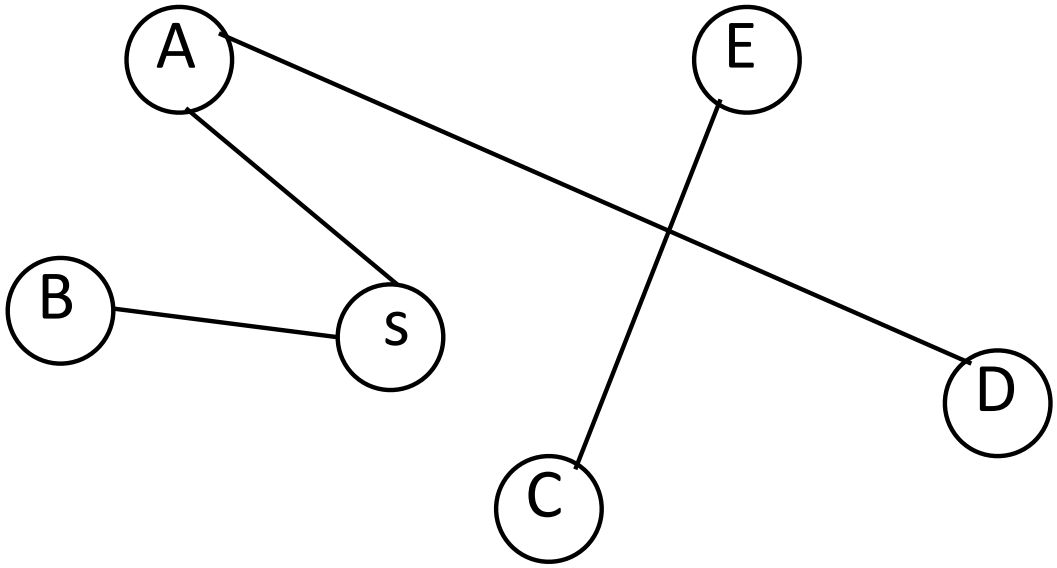
# Question: exploreDFS

Which vertices does exploreDFS(s) mark as visited?

A. All the vertices

B. All vertices except C & E

C. None of the above



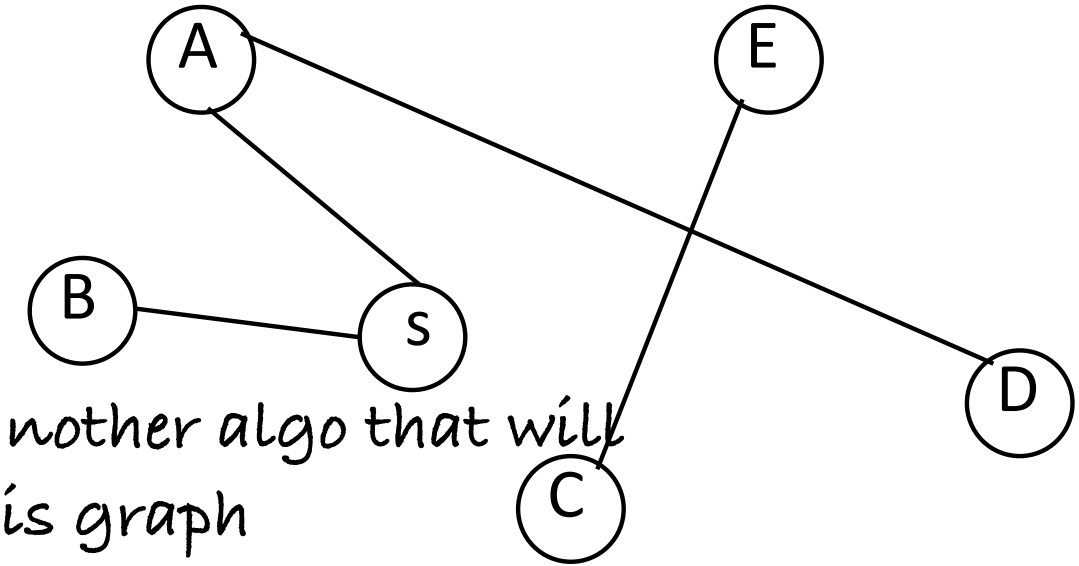
# Question: exploreDFS

Which vertices does exploreDFS(s) mark as visited?

A. All the vertices

B. All vertices except C & E

C. None of the above




Use exploreDFS to write another algo that will visit all the vertices in this graph

# Depth First Search

`exploreDFS` only finds the part of the graph reachable from a single vertex. If you want to discover the entire graph, you may need to run it multiple times.

visited

0, 2, 4, 5

1, 3

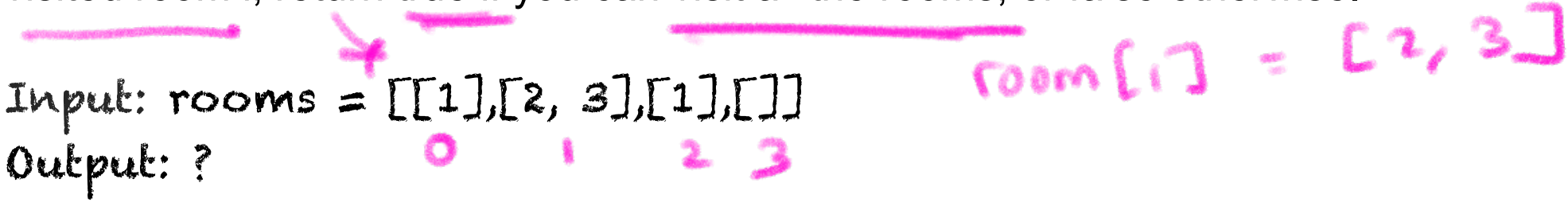
0

```
DepthFirstSearch(G)

  Mark all v ∈ G as unvisited

  For v ∈ G

    If not v.visited, exploreDFS(v)
```

There are n rooms labeled from 0 to n - 1 and all the rooms are locked except for room 0. Your goal is to visit all the rooms. However, you cannot enter a locked room without having its key.

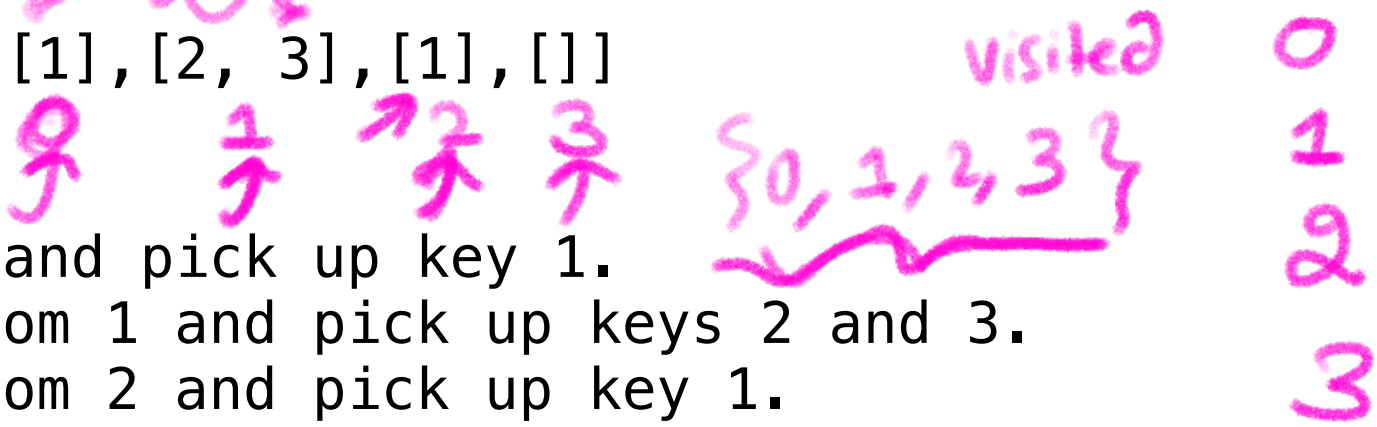
When you visit a room, you may find a set of distinct keys in it. Each key has a number on it, denoting which room it unlocks, and you can take all of them with you to unlock the other rooms.

Given an array rooms where rooms[i] is the set of keys that you can obtain if you visited room i, return true if you can visit all the rooms, or false otherwise.




https://leetcode.com/problems/keys-and-rooms/description/

**Input:** rooms = [[1],[2, 3],[1],[]]
**Output:** true
**Explanation:**
We visit room 0 and pick up key 1.
We then visit room 1 and pick up keys 2 and 3.
We then visit room 2 and pick up key 1.
We then visit room 3.
Since we were able to visit every room, we return true.




https://leetcode.com/problems/keys-and-rooms/description/

One approach proposed was to iterate over the rooms vector and insert all keys in each rooms[i] into a hashtable. Return true if the size of the hashtable == number of rooms

The above approach works for this example input

rooms = [[1], [2,3], [], []]
0 1 2 3

unordered_set = {0, 1, 2, 3} returns true

However it will not work for the following example (expected true)

rooms = [[1], [], [1,2,3], []]
0 1 2 3

unordered_set = {0, 1, 2, 3} | Suggested approach returns true, expected false

# Cast as a graph problem

adjacency list!

```
Input:  vector<vector<int>> rooms = [[1],[2, 3],[1],[]]
Output: true
```

0 1 2 3

Discuss with your peer (5 mins):

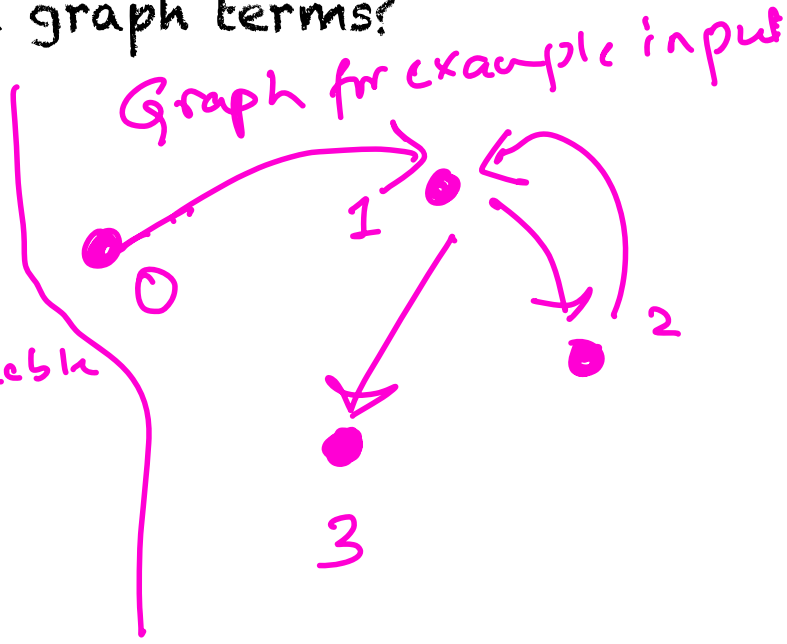
1. What are the nodes and what are the edges in this problem?
2. What does the input (rooms) represent in graph terms?

$V = \{0, 1, 2, 3\}$

Edge (i, j) means i has a key to unlock j.

≡ Room j is reachable from i

0 → [1]
1 → [2, 3]
2 → [1]
3 → []



Approach: Perform Depth First / Breadth First Traversal on the graph representing the problem

return true if all rooms could be visited.
false otherwise.

We wrote the code for this problem in lecture