<https://leetcode.com/problems/keys-and-rooms/>

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*.

**Example 1:**

Input: rooms = [[1],[2],[3],[]]

Output: true

Explanation:

We visit room 0 and pick up key 1.

We then visit room 1 and pick up key 2.

We then visit room 2 and pick up key 3.

We then visit room 3.

Since we were able to visit every room, we return true.

**Example 2:**

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

Output: false

Explanation: We can not enter room number 2 since the only key that unlocks it is in that room.

**Constraints:**

n == rooms.length

2 <= n <= 1000

0 <= rooms[i].length <= 1000

1 <= sum(rooms[i].length) <= 3000

0 <= rooms[i][j] < n

All the values of rooms[i] are **unique**.

**Attempt 1: 2022-11-11**

**Solution 1:  Recursive traversal as DFS (10min)**

class Solution {

    Set<Integer> visited;

    public boolean canVisitAllRooms(List<List<Integer>> rooms) {

        visited = new HashSet<Integer>();

        // Start with room 0

        visited.add(0);

        helper(0, rooms);

        return visited.size() == rooms.size();

    }

    private void helper(int roomId, List<List<Integer>> rooms) {

        List<Integer> pendingVisitRooms = rooms.get(roomId);

        for(int i : pendingVisitRooms) {

            // Only go into unvisited room, otherwise circle path dead loop

            if(!visited.contains(i)) {

                visited.add(i);

                helper(i, rooms);

            }

        }

    }

}

Time Complexity: O(N + E), where N is the number of rooms and E the number of total keys (edges)

Doing O(1) work for all N rooms - checking if seen, adding to queue/stack

Doing O(1) work for all E keys - going through the list of neighbors for each room

Space Complexity: O(N)

visited set is O(N) as its keeping track of all the vertices, data structure used to process vertices

**Solution 2:  Iterative traversal  as BFS (10min)**

class Solution {

    public boolean canVisitAllRooms(List<List<Integer>> rooms) {

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

        Queue<Integer> q = new LinkedList<Integer>();

        // Start with room 0

        q.offer(0);

        visited.add(0);

        while(!q.isEmpty()) {

            int size = q.size();

            for(int i = 0; i < size; i++) {

                int roomId = q.poll();

                List<Integer> pendingVisitRooms = rooms.get(roomId);

                for(int j : pendingVisitRooms) {

                    // Only go into unvisited room, otherwise circle path dead loop

                    if(!visited.contains(j)) {

                        visited.add(j);

                        q.offer(j);

                    }

                }

            }

        }

        return visited.size() == rooms.size();

    }

}

Time Complexity: O(N), where N is the number of rooms

Space Complexity: O(N)

**Solution 3: Check if Directed Graph is strongly connected or not (10min)**

class Solution {

    public boolean canVisitAllRooms(List<List<Integer>> rooms) {

        // Build graph

        Map<Integer, List<Integer>> adjacencyList = new HashMap<Integer, List<Integer>>();

        for(int i = 0; i < rooms.size(); i++) {

            adjacencyList.put(i, new ArrayList<Integer>());

            for(int j = 0; j < rooms.get(i).size(); j++) {

                adjacencyList.get(i).add(rooms.get(i).get(j));

            }

        }

        // Recursion check

        return isConnected(adjacencyList);

    }

    private boolean isConnected(Map<Integer, List<Integer>> adjacencyList) {

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

// Perform DFS from the starting node (room 0) as required by problem

// Note: if we strictly follow the definition of "DFS check on directed

// graph connected (strongly connected) or not" case, we should attempt

// with all rooms (Refer the document "Directed and Undirected Graph

// Connectedness"), but since the problem limits only start with room 0,

// which degenerates the standard directed graph connection problem into

// a single start vertex directed graph connection problem, more like

// undirected graph behavior which requires only randomly pick one start

        helper(0, adjacencyList, visited);

        return visited.size() == adjacencyList.size();

    }

    private void helper(int startRoomId, Map<Integer, List<Integer>> adjacencyList, Set<Integer> visited) {

        visited.add(startRoomId);

        List<Integer> adjacencies = adjacencyList.get(startRoomId);

        for(int i : adjacencies) {

            if(!visited.contains(i)) {

                helper(i, adjacencyList, visited);

            }

        }

    }

}

Time Complexity: O(N + E), where N is the number of rooms and E the number of total keys (edges)

Doing O(1) work for all N rooms - checking if seen, adding to queue/stack

Doing O(1) work for all E keys - going through the list of neighbors for each room

Space Complexity: O(N)

visited set is O(N) as its keeping track of all the vertices, data structure used to process vertices

**Why "Keys and Rooms" problem recognize as a Directed Graph problem ?**

**Refer to**

<https://leetcode.com/problems/keys-and-rooms/discuss/1756839/Python-or-Simple-DFS-%2B-BFS-or-Explained-w-Complexity>

**Intuition**

The problem statement tell us that we have access to a room y only if the key is a previous room x. This highlights a specific, directional relationship: x --> y. Relationships between objects can naturally be viewed through a graph lens. We can put it in a graph context by viewing the rooms as vertices in a **directed graph**. More formally, we know that there is an edge (u,v) if vertex v is in the list rooms[u], i.e. if the room u contains a key to room v. We are trying to determine whether we can unlock the entire set of rooms (i.e. visit all the nodes in the graph) if we start at room (node) 0. We can thus reduce this problem to running a graph traversal and, at its conclusion, checking to see if we have indeed seen all n rooms.

**Approach**

The type of traversal doesn't matter - both BFS and DFS do the job. We keep track of the rooms we've seen via an array seen. seen[i] indicates whether we have unlocked room i. At the end of the traversal, if there is any index in seen that contains a False value, we know that we were not able to unlock all of the rooms.

**Implementation**

Code for DFS:

class Solution:

    def canVisitAllRooms(self, rooms: List[List[int]]) -> bool:

        n = len(rooms)

        seen = [False] \* n

        stack = [0]  # room 0 is unlocked

        while stack:

            room = stack.pop()

            if not seen[room]:  # if not previously visited

                seen[room] = True

                for key in rooms[room]:

                        stack.append(key)

        return all(seen)

Code for BFS:

from collections import deque

class Solution:

    def canVisitAllRooms(self, rooms: List[List[int]]) -> bool:

        n = len(rooms)

        seen = [False] \* n

        queue = deque([0]) # room 0 is unlocked

        while queue:

            room = queue.popleft()

            seen[room] = True

            for key in rooms[room]:

                if not seen[key]:  # if not previously visited

                    queue.append(key)

                    seen[key] = True

        return all(seen)

**Complexity**

Time: O(N + E), where N is the number of rooms and E the number of total keys (edges)

Doing O(1) work for all N rooms - checking if seen, adding to queue/stack

Doing O(1) work for all E keys - going through the list of neighbors for each room

Space: O(N)

seen array is O(N) as its keeping track of all the vertices

data structure used to process vertices (queue or stack depending on traversal) will hold at most O(N) elements at once

**Why it looks like a DFS check undirected graph connected or not ?**

**Perform DFS from the starting node (room 0) as required by problem**

**Note: if we strictly follow the definition of "DFS check on directed graph connected (strongly connected) or not" case, we should attempt with all rooms (Refer the document** [Directed and Undirected Graph Connectedness](note://703FB1A74E454327B7F34E47F95EDAB5)**), but since the problem limits only start with room 0, which degenerates the standard directed graph connection problem into a single start vertex directed graph connection problem, more like undirected graph behavior which requires only randomly pick one start**

**Refer to**

<https://leetcode.com/problems/keys-and-rooms/discuss/135306/BFS-(9-lines-10ms)-and-DFS-(7-lines-18ms)-in-C++-w-beginner-friendly-explanation/183673>

The question can be viewed as a simple directed graph question: IS THE DIRECTED GRAPH CONNECTED? ( i.e. Only one Connected Component, start requires on room 0)

Here is the long explanatory solution of the same

class Solution {

public:

    void dfs(unordered\_map<int, vector<int> > &adjacencyList, int v1, unordered\_map<int, bool> &visited)

    {

      vector<int> adjacentVertices = adjacencyList[v1];

      // Mark it visited to avoid calling over this vertex again

      visited[v1] = true;

      // Recursion call

      for(int i = 0; i < adjacentVertices.size(); i++)

      {

        //If the vertex is not visited yet then dfs from there

        if(visited.find(adjacentVertices[i]) == visited.end())

          dfs(adjacencyList, adjacentVertices[i], visited);

      }

    }

    bool isConnected(unordered\_map<int, vector<int> > &adjacencyList)

    {

      unordered\_map<int, bool> visited;

      int startVertex = 0;

      dfs(adjacencyList, startVertex, visited);

      return visited.size() == adjacencyList.size();

    }

    /\* Main Function \*/

    bool canVisitAllRooms(vector<vector<int>>& rooms)

    {

        unordered\_map<int, vector<int> > adjacencyList;

        //Putting the input in a graph

        for(int i=0;i<rooms.size();i++)

        {

            //Putting the room in map irrespective it has any keys or not

            adjacencyList[i];

            for(auto &r: rooms[i])

                adjacencyList[i].push\_back(r);

        }

        return isConnected(adjacencyList) ;

    }

};

**Java version (convert by chatGPT)**

class Solution {

    public void dfs(Map<Integer, List<Integer>> adjacencyList, int v1, Map<Integer, Boolean> visited) {

        List<Integer> adjacentVertices = adjacencyList.get(v1);

        // Mark it visited to avoid calling over this vertex again

        visited.put(v1, true);

        // Recursion call

        for (int i = 0; i < adjacentVertices.size(); i++) {

            // If the vertex is not visited yet then dfs from there

            if (!visited.containsKey(adjacentVertices.get(i))) {

                dfs(adjacencyList, adjacentVertices.get(i), visited);

            }

        }

    }

    public boolean isConnected(Map<Integer, List<Integer>> adjacencyList) {

        Map<Integer, Boolean> visited = new HashMap<>();

        int startVertex = 0;

        dfs(adjacencyList, startVertex, visited);

        return visited.size() == adjacencyList.size();

    }

    /\* Main Function \*/

    public boolean canVisitAllRooms(List<List<Integer>> rooms) {

        Map<Integer, List<Integer>> adjacencyList = new HashMap<>();

        // Putting the input in a graph

        for (int i = 0; i < rooms.size(); i++) {

            // Putting the room in map irrespective it has any keys or not

            adjacencyList.put(i, new ArrayList<>());

            for (Integer r : rooms.get(i))

                adjacencyList.get(i).add(r);

        }

        return isConnected(adjacencyList);

    }

}

**Refer to**

[L261.Lint178.Graph Valid Tree (Ref.L841)](note://2DB801CF2BAB4DDE9E74C819757178B5)

[Directed and Undirected Graph Connectedness](note://703FB1A74E454327B7F34E47F95EDAB5)