841. Keys and Rooms

Medium | Depth-First Search | Breadth-First Search | Graph

Problem Description

Imagine you're in the first room of a series of locked rooms, each with a unique label from `0` to `n - 1`, where `n` is the total number of rooms. Room `0` is unlocked, so you can start your exploration from there. Each room may contain a bunch of keys, and each key unlocks exactly one specific room. Your objective is to figure out if you can manage to enter every single room. The twist is, you can't waltz into a locked room without its corresponding key, which you would typically find in one of the other rooms you can access.

The question presents this scenario in the form of an array called `rooms`. This array is structured so that `rooms[i]` represents a list of keys that you can pick up when you enter room `i`. You must determine if it's possible for you to gather all the necessary keys from the rooms you can access to ultimately open all rooms.

To solve this, you'll need to figure out a way to traverse through the rooms, beginning from the starting room (room `0`), while keeping track of the rooms you've entered and the keys you've collected along the way.

Intuition

To solve this puzzle, we need to think like an adventurer who's exploring a dungeon. Starting from the entrance, you go into each room, gather all the keys you find, and keep track of the rooms you've visited. Every time you find a new key, you unlock a new room and explore it for more keys.

Translating this into a solution, a Depth-First Search (DFS) approach naturally fits the problem. What we need to do is akin to exploring a graph where rooms are nodes and keys are edges connecting the nodes. We start from room `0` and try to reach all other rooms using the keys we collect.

Here's the thought process that leads to using DFS:

1. Start from the initial room, which is room `0`.
2. If a room has been visited or a key has been used, there is no need to re-visit or re-use it.
3. Traversal continues until there are no reachable rooms with new keys left.
4. If at the end of the traversal, the number of visited rooms is equal to the total number of rooms, it means we've successfully visited all rooms.
5. If there are still rooms left unvisited, it implies there's no way to unlock them with the keys available, hence the answer would be false.

The solution provided uses a recursive DFS function to perform this traversal, calling upon itself each time a new room is reached. The set `vis` helps track visited rooms, ensuring rooms are not revisited. The return value checks if the size of `vis` matches the total number of rooms.

Solution Approach

To implement the solution for unlocking all rooms, we use a Depth-First Search (DFS) algorithm. DFS is a common algorithm used to traverse graphs, which fits our scenario perfectly as the problem can be conceptualized as a graph where rooms are nodes and keys are the edges that connect these nodes.

Here's how the implemented DFS algorithm works step-by-step:

1. We define a recursive function `dfs(u)` which accepts a room number `u` as a parameter.
2. The function begins by checking if the room `u` has already been visited. To efficiently check for previously visited rooms, we make use of a set called `vis`. If the current room `u` is in this set, we quit the function early since we've already been here.
3. If the room hasn't been visited, we add the room number `u` to the set `vis`. This marks room `u` as visited and ensures we don't waste time revisiting it in future DFS calls.
4. Next, we iterate over all the keys found in room `u`, which are represented by `rooms[u]`.
5. For each key `v` in `rooms[u]`, we perform a recursive DFS call with `v`. This is the exploration step where we are trying to dive deeper into the graph from the current node (room).
6. We start our DFS by calling `dfs(0)` — since room `0` is where our journey begins, and it is the first room that is unlocked.
7. After fully exploring all reachable rooms, we check if we have visited all rooms. To do this, we compare the length of our visited set `vis` with the length of the `rooms` array. If they match, it means we have successfully visited every room.

The data structure used in this implementation is primarily a set (`vis`). Sets in Python are collections that are unordered, changeable, and do not allow duplicate elements. The characteristics of a set make it an ideal choice to keep track of visited rooms because it provides O(1) complexity for checking the presence of an element, which is crucial for an efficient DFS implementation.

By utilizing recursion for the depth-first traversal, and a set to track the visited nodes, we apply a classical graph traversal technique to solve this problem. It's concise, efficient, and directly addresses our goal of determining whether all rooms can be visited.

By running the DFS from room `0` and continually marking each room visited and exploring the next accessible rooms through the keys, the `canVisitAllRooms` function ultimately returns `True` if every room has been reached, and `False` if at least one room remains locked after the search is complete.

Example Walkthrough

Let's consider a small set of rooms to illustrate the solution approach:

Suppose there are `4` rooms, and the keys are distributed as follows:

- Room `0` contains keys for rooms `1` and `2`.
- Room `1` contains a key for room `3`.
- Room `2` contains no keys.
- Room `3` contains no keys.

Representing this in an array, we have rooms = `[[1,2], [3], [], []]`.

Using the DFS approach:

1. We start the DFS with `dfs(0)`, marking room `0` as visited and adding it to the set `vis`.
2. We find two keys in room `0`, which are to rooms `1` and `2`, so we will traverse to these rooms next.
3. First, we call `dfs(1)`. Before diving into room `1`, we add it to the `vis` set.
4. Inside room `1`, we find a key to room `3`. We now call `dfs(3)` since room `3` hasn't been visited yet.
5. We add room `3` to the `vis` set, but find no keys in it.
6. With no further keys from room `3`, we backtrack to room `1`, and since we've explored all its keys, we backtrack further to room `0`.
7. Now, we call `dfs(2)` from the remaining key in room `0`.
8. Room `2` is added to the `vis` set, but since there are no keys, we have nothing further to explore from here.
9. We have now visited all the rooms: `vis = {0, 1, 2, 3}`.
10. We compare the number of visited rooms with the total number of rooms. Since both are `4`, we have successfully visited every room.

By following each key we find to the next room and keeping track of where we've been, we confirmed that it is possible to visit all the rooms. In the scenario presented by `rooms`, our implementation of the `canVisitAllRooms` function will return `True`.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def canVisitAllRooms(self, rooms: List[List[int]]) -> bool:
        # Depth-first search (DFS) function that marks rooms as visited
        def dfs(current_room):
            # If the current room has already been visited, do nothing
            if current_room in visited:
                return
            # Mark the current room as visited
            visited.add(current_room)
            # Perform a DFS on all the keys/rooms that are accessible from the current room
            for key in rooms[current_room]:
                dfs(key)

        # A set to keep track of visited rooms
        visited = set()
        # Initiate DFS starting from room 0
        dfs(0)
        # If the number of visited rooms equals the total number of rooms, all rooms can be visited
        return len(visited) == len(rooms)
```

Java

```java
import java.util.List;
import java.util.Set;
import java.util.HashSet;

class Solution {
    private List<List<Integer>> rooms;  // A list representing the keys in each room.
    private Set<Integer> visited;       // A set to track visited rooms.

    // Method to check if we can visit all rooms starting from room 0.
    public boolean canVisitAllRooms(List<List<Integer>> rooms) {
        this.rooms = rooms; // Initialize the room list.
        visited = new HashSet<>(); // Initialize visited set.
        dfs(0); // Start depth-first search from room 0.
        // If the size of visited rooms is equal to total room count, return true.
        return visited.size() == rooms.size();
    }

    // Recursive method for depth-first search.
    private void dfs(int roomIndex) {
        // If the room has already been visited, return to avoid cycles.
        if (visited.contains(roomIndex)) {
            return;
        }
        // Mark the current room as visited.
        visited.add(roomIndex);
        // Recursively visit all rooms that can be unlocked with keys from the current room.
        for (int nextRoomIndex : rooms.get(roomIndex)) {
            dfs(nextRoomIndex);
        }
    }
}
```

C++

```cpp
#include <vector>
#include <unordered_set>

class Solution {
public:
    // Stores the list of rooms and keys within them
    std::vector<std::vector<int>> roomKeys;

    // Keeps track of visited rooms
    std::unordered_set<int> visitedRooms;

    // Returns true if all rooms can be visited using the keys in them
    bool canVisitAllRooms(std::vector<std::vector<int>>& rooms) {
        // Clear the visitedRooms set in case it's being reused
        visitedRooms.clear();

        // Initialize roomKeys with the input rooms and their keys
        roomKeys = rooms;

        // Start the DFS from room 0
        dfs(0);

        // If the size of visitedRooms is equal to the total number of rooms,
        // it means we could visit all rooms
        return visitedRooms.size() == rooms.size();
    }

    // Recursive Depth First Search function to visit rooms
    void dfs(int currentRoom) {
        // If we've already visited the current room, return to avoid cycles
        if (visitedRooms.count(currentRoom)) return;

        // Mark the current room as visited
        visitedRooms.insert(currentRoom);

        // Loop over every key in the current room
        for (int key : roomKeys[currentRoom]) {
            // Use the key to visit the next room
            dfs(key);
        }
    }
};
```

TypeScript

```typescript
// Define a function to determine if all rooms can be visited given a list of rooms and their keys.
// Each room is represented by an index and contains a list of keys to other rooms.
function canVisitAllRooms(rooms: number[][]): boolean {
    // Determine the total number of rooms.
    const totalRooms = rooms.length;

    // Create an array to keep track of whether each room has been opened.
    const isRoomOpen = new Array(totalRooms).fill(false);

    // Initialize a stack with the key to the first room (i.e., room 0).
    const keysStack = [0];

    // Process keys while there are still keys left in the stack.
    while (keysStack.length !== 0) {
        // Retrieve the last key from the stack.
        const currentKey = keysStack.pop();

        // Skip processing if the room has already been opened.
        if (isRoomOpen[currentKey]) {
            continue;
        }

        // Mark the current room as opened.
        isRoomOpen[currentKey] = true;

        // Add all the keys found in the current room to the stack.
        keysStack.push(...rooms[currentKey]);
    }

    // Check if all rooms have been opened. If so, return true.
    return isRoomOpen.every(isOpen => isOpen);
}
```


```python
from typing import List

class Solution:
    def canVisitAllRooms(self, rooms: List[List[int]]) -> bool:
        # Depth-first search (DFS) function that marks rooms as visited
        def dfs(current_room):
            # If the current room has already been visited, do nothing
            if current_room in visited:
                return
            # Mark the current room as visited
            visited.add(current_room)
            # Perform a DFS on all the keys/rooms that are accessible from the current room
            for key in rooms[current_room]:
                dfs(key)

        # A set to keep track of visited rooms
        visited = set()
        # Initiate DFS starting from room 0
        dfs(0)
        # If the number of visited rooms equals the total number of rooms, all rooms can be visited
        return len(visited) == len(rooms)
```


Time and Space Complexity

The time complexity of the code is `O(N + E)`, where `N` represents the number of rooms and `E` represents the total number of keys (or edges in graph terms, representing connections between rooms). Each room and key is visited exactly once due to the depth-first search (DFS) algorithm used, and the `vis` set ensures that each room is entered only once.

The space complexity of the code is also `O(N)` because the `vis` set can potentially contain all `N` rooms at the worst-case scenario when all rooms are visited. Additionally, the recursive stack due to DFS also contributes to the space complexity, and in the worst case might store all the rooms if they are connected linearly, making the space complexity still `O(N)`.