1997. First Day Where You Have Been in All the Rooms

Medium | Array | Dynamic Programming

Problem Description

The task is to determine the first day on which you have visited all the rooms in a sequence of `n` rooms. The rooms are labeled from `0` to `n-1`, and each day is also labeled starting from `0`. On the first day (day `0`), you visit room `0`. The order in which you visit the rooms on subsequent days is determined by two rules and an array `nextVisit`:

- If you are visiting a room `i` for an odd number of times, the next room you will visit is the one with the number `nextVisit[i]` (where `0 <= nextVisit[i] <= i`).
- If you are visiting room `i` for an even number of times, the next room you will visit is `(i + 1) mod n`.

The goal is to return the label of the first day (mod `10^9 + 7`) where you have been in all the rooms at least once. The problem guarantees there is always such a day.

Intuition

The solution is based on dynamic programming. To find the first day when all rooms have been visited, an array `f` is used to store the number of days needed to visit all rooms up to index `i`.

- For each room `i` after the first (since we start in room `0` on day `0`), there are a certain number of steps needed to get to room `i` from room `0`.
- Upon entering room `i` for the first time, which is always an odd visit, you must follow the `nextVisit[i-1]` to determine the next room. Then, you will return to room `i` and go to `i+1` on the even visit.
- The formula `f[i] = (f[i - 1] + 1 + f[i - 1] - f[nextVisit[i - 1]] + 1) % mod` encapsulates this behavior. It calculates the total days taken to reach room `i` by adding:
  - The days to reach the previous room, `f[i - 1]`, plus one day to go to `nextVisit[i-1]` room following the odd visit rule,
  - The days to return from `nextVisit[i-1]` to room `i` again, which is `f[i - 1] - f[nextVisit[i - 1]]` plus one day for the even visit to move to the next room.
- This process is repeated until the days needed to reach the last room are determined. The value of `f[-1]` gives the first day when all rooms have been visited, which is taken modulo `10^9 + 7` to keep the number within the integer limits.

The solution leverages the overlapping subproblems characteristic of dynamic programming where the result of visiting previous rooms is reused to determine the number of days needed to reach the current room.

Solution Approach

The reference solution implements a dynamic programming approach to solve the problem. Here's a walk-through of the implementation and the concepts used:

1. **Initialization**: We initialize a list `f` of length `n` (where `n` is the total number of rooms), with all values set to `0`. This list will hold the minimum number of days required to reach each room for the first time. The room `0` is already visited on day `0`, hence `f[0]` is initially `0`.

2. **Modulo Constant**: We define `mod` as `10**9 + 7` which is used for taking modulo after calculations to prevent integer overflow and as required by the problem statement.

3. **Dynamic Programming Iteration**: For each room `i` starting from `1` to `n-1` (since room `0` is the starting room and does not require computation), we calculate the minimum number of days to reach this room for the first time, denoted by `f[i]`. The formula used is:

```
f[i] = (f[i - 1] + 1 + f[i - 1] - f[nextVisit[i - 1]] + 1) % mod
```

   The terms in the formula have specific meanings:

   - `f[i - 1]`: Minimum days to reach the previous room (`i-1`).
   - `f[i - 1] - f[nextVisit[i - 1]]`: Days spent to visit `nextVisit[i - 1]` and return to room `i`. This is the difference of days needed to reach the previous room `i-1` and days needed to reach `nextVisit[i - 1]`.
   - The two `+1`s in the formula account for the days spent for the odd visit (to move to `nextVisit[i - 1]`) and the even visit (to return to room `i` and proceed to the next room).
4. **Result**: After iterating through all rooms, `f[-1]` (the last element of `f`) will contain the first day where all rooms have been visited at least once. This value accounts for all the previous visits and follows the given rules.

5. **Time Complexity**: The time complexity of the solution is `O(n)` since it iterates over the `n` rooms once to fill out the dynamic programming table `f`.

6. **Space Complexity**: Since only one additional array `f` of size `n` is used, the space complexity of the algorithm is also `O(n)`.

By applying this dynamic programming approach, we leverage previous computations to efficiently find the target day, as each step's result depends on the plurality of previous steps but avoids redundant recomputation of those steps.

Example Walkthrough

Let's consider a small example where `n = 4` and the array `nextVisit` is given by `[0, 1, 2, 0]`. This means we have 4 rooms in total and the next room you will visit on an odd occasion is determined by the values in `nextVisit`.

1. **Initialization**: We have `f = [0, 0, 0, 0]` because we have 4 rooms and room `0` is visited on day `0`, so `f[0]` is `0` by default.

2. **First Calculation (for room 1)**:

   - We use the formula `f[i] = (f[i - 1] + 1 + f[i - 1] - f[nextVisit[i - 1]] + 1) % mod`.
   - For room `1`, `i = 1`, `nextVisit[0] = 0`, so we get:
     - `f[1] = (f[0] + 1 + f[0] - f[nextVisit[0]] + 1) % mod = (0 + 1 + 0 - 0 + 1) % mod = 2`.
   - This means we will reach room `1` for the first time on day `2`.
3. **Second Calculation (for room 2)**:

   - For room `2`, `i = 2`, `nextVisit[1] = 1`, we apply the formula:
     - `f[2] = (f[1] + 1 + f[1] - f[nextVisit[1]] + 1) % mod = (2 + 1 + 2 - 2 + 1) % mod = 4`.
   - This means we will reach room `2` for the first time on day `4`.
4. **Third Calculation (for room 3)**:

   - For room `3`, `i = 3`, `nextVisit[2] = 2`, we apply the formula:
     - `f[3] = (f[2] + 1 + f[2] - f[nextVisit[2]] + 1) % mod = (4 + 1 + 4 - 4 + 1) % mod = 6`.
   - This means we will reach room `3` for the first time on day `6`.
5. **Final Step**: Now, finally, our dynamic array `f` is `[0, 2, 4, 6]`, and since we are asked to find the first day when we have been in all the rooms at least once, our answer is `f[-1]`, which is `6`.

This example clearly demonstrates how the dynamic programming approach builds on the solution to previous rooms to efficiently calculate the arrival at each subsequent room. Here, the array `f` tracks the day on which each room is first reached using the calculated formula, adhering accurately to the sequence of visits dictated by the `nextVisit` array.

Solution Implementation

Python

```python
class Solution:
    def firstDayBeenInAllRooms(self, nextVisit: List[int]) -> int:
        # Get the total number of rooms to visit
        number_of_rooms = len(nextVisit)

        # Initialize an array to store the days to reach each room for the first time
        days_to_reach = [0] * number_of_rooms

        # Define the modulo for large number handling (to prevent integer overflow)
        mod = 10**9 + 7

        # Iterate over the rooms starting from the second room, as the first room's day count is zero by default
        for i in range(1, number_of_rooms):
            # Calculate number of days to reach this room for the first time
            # The formula is based on the previous room's day count and the day count at the
            # index of the nextVisit for the previous room. We visit the current room after
            # visiting the previous room twice and once after nextVisit for the previous room.
            days_to_reach[i] = (days_to_reach[i - 1] + 1 + days_to_reach[i - 1] - days_to_reach[nextVisit[i - 1]] + 1) % mod

        # Return the number of days to reach the last room for the first time
        return days_to_reach[-1]
```

Java

```java
class Solution {
    public int firstDayBeenInAllRooms(int[] nextVisit) {
        // Get the number of rooms, which is also the length of the input array.
        int numRooms = nextVisit.length;
        // Create an array to store the number of days taken to reach each room for the first time.
        long[] daysToEnterRoom = new long[numRooms];
        // Define the modulo constant as per the problem statement (1e9 + 7).
        final int MODULO = (int) 1e9 + 7;

        // Iterate over each room starting from the second room (since the first room is the starting point).
        for (int i = 1; i < numRooms; ++i) {
            // Calculate the number of days to reach the current room for the first time.
            // It is based on the days to enter the previous room, plus one day to move to the next current room, plus
            // the number of days to re-enter the current room after visiting the 'nextVisit' room.
            // The additional modulo operation ensures that the number stays within the integer range.
            daysToEnterRoom[i] = (daysToEnterRoom[i - 1] + 1 + daysToEnterRoom[i - 1] - daysToEnterRoom[nextVisit[i - 1]] + 1 + M
        }

        // Return the number of days taken to reach the last room for the first time.
        return (int) daysToEnterRoom[numRooms - 1];
    }
}
```

C++

```cpp
class Solution {
public:
    int firstDayBeenInAllRooms(vector<int>& nextVisit) {
        int numRooms = nextVisit.size(); // Store the number of rooms
        vector<long long> daysToVisit(numRooms); // Create a vector to keep track of the days needed to visit each room
        const int MOD = 1e9 + 7; // Define the modulo value for large numbers to prevent integer overflow

        // Loop through each room starting from the second room since the first room is always visited on day 0
        for (int i = 1; i < numRooms; ++i) {
            // The days to visit current room 'i' is equal to:
            // Days to visit the previous room + 1 (for today's visit) +
            // Days to visit the previous room again after the extra day of waiting,
            // plus 2 more days (as you visit nextVisit[i - 1] and then move to room 'i') -
            // Days to visit the room pointed to by nextVisit[i - 1].
            // We also add MOD before taking the modulo to handle any negative numbers.
            daysToVisit[i] = (daysToVisit[i - 1] + 1 + daysToVisit[i - 1] - daysToVisit[nextVisit[i - 1]] + 2 + MOD) % MOD;
        }

        // Return the total days needed to visit the last room
        return daysToVisit[numRooms - 1];
    }
};
```

TypeScript

```typescript
const MOD = 1e9 + 7;  // Define the modulo constant to prevent integer overflow

function firstDayBeenInAllRooms(nextVisit: number[]): number {
    const numRooms = nextVisit.length;  // Store the number of rooms
    let daysToVisit: number[] = new Array(numRooms).fill(0);  // Create an array to keep track of the days needed to visit each

    // Iterate through each room starting from the second room, since the first room is always visited on day 0
    for (let i = 1; i < numRooms; i++) {
        // Calculate the days to visit the current room 'i':
        // Add one for visiting the previous room, plus the days to reach the previous room again after the revisit,
        // plus two more days (for visiting the room at nextVisit[i - 1] and then moving to room 'i').
        // Subtract the days to visit the room pointed to by nextVisit[i - 1].
        // MOD is added before taking the modulo to handle any negative numbers.
        daysToVisit[i] = (daysToVisit[i - 1] + 1 + daysToVisit[i - 1] + 2 - daysToVisit[nextVisit[i - 1]] + MOD) % MOD;
    }

    // Return the total days needed to visit the last room
    return daysToVisit[numRooms - 1];
}

// Example usage
// const nextVisit = [0,0]; // An example input for the function
// console.log(firstDayBeenInAllRooms(nextVisit)); // Call the function and log the output
```

```python
class Solution:
    def firstDayBeenInAllRooms(self, nextVisit: List[int]) -> int:
        # Get the total number of rooms to visit
        number_of_rooms = len(nextVisit)

        # Initialize an array to store the days to reach each room for the first time
        days_to_reach = [0] * number_of_rooms

        # Define the modulo for large number handling (to prevent integer overflow)
        mod = 10**9 + 7

        # Iterate over the rooms starting from the second room, as the first room's day count is zero by default
        for i in range(1, number_of_rooms):
            # Calculate number of days to reach this room for the first time
            # The formula is based on the previous room's day count and the day count at the
            # index of the nextVisit for the previous room. We visit the current room after
            # visiting the previous room twice and once after nextVisit for the previous room.
            days_to_reach[i] = (days_to_reach[i - 1] + 1 + days_to_reach[i - 1] - days_to_reach[nextVisit[i - 1]] + 1) % mod

        # Return the number of days to reach the last room for the first time
        return days_to_reach[-1]
```

Time and Space Complexity

The provided code defines a function `firstDayBeenInAllRooms` which calculates the first day a person has been in all rooms given a sequence of `nextVisit` specifying the next room to visit.

The time complexity of `firstDayBeenInAllRooms` is `O(n)`, where `n` is the length of the input list `nextVisit`. This is because there is a single loop that iterates through each room exactly once. Each iteration involves constant-time arithmetic operations and a modulo operation, which do not depend on the size of the input.

The space complexity of the function is also `O(n)` because it allocates an array `f` of size `n` to store the number of days taken to reach every room until the last one. No other data structures are used that scale with the input size, hence the space complexity is linear with respect to the input size.

In the loop, for each `i` (room), the following formula is used to calculate `f[i]`, which represents the first day the person has been in all rooms up to room `i`:

```
f[i] = (f[i - 1] + 1 + f[i - 1] - f[nextVisit[i - 1]] + 1) % mod
```

The modulo operation with `mod = 10**9 + 7` ensures that the result stays within the bounds of a 32-bit integer, which is a common practice to avoid overflow in programming contests and problems.