2327. Number of People Aware of a Secret

Medium | Queue | Dynamic Programming | Simulation | Leetcode Link

Problem Description

This LeetCode problem presents a scenario in which a secret spreads among people according to specific rules. On the first day, one person discovers a secret. Two parameters, `delay` and `forget`, govern how the secret spreads:

- `delay`: The number of days after a person discovers the secret before they begin sharing it every day.
- `forget`: The number of days after discovering the secret when a person forgets it and stops sharing.

The goal is to calculate how many people know the secret at the end of `n` days. To handle large numbers, the result should be returned modulo `10^9 + 7`, a common practice to prevent integer overflow in programming contests.

Intuition

The solution strategy involves simulating the sharing and forgetting process over `n` days, while efficiently tracking the count of people who know the secret and can still share it. The key steps are:

1. **Tracking the Spread and Forgetting**: The number of people who can share the secret changes over time and needs to be accurately tracked each day, from the day they start sharing to the day they forget the secret.

2. **Cumulative Counting**: Instead of updating individual counts every day, the solution uses a cumulative count approach to record changes in a manner that allows for all future updates at once.

3. **Daily Update**: The code simulates each day, updating the count of people who will share and forget the secret in the future.

4. **Modular Arithmetic**: Since the final answer can be very large, it's necessary to use modular arithmetic to keep the numbers within bounds and compute the result modulo `10^9 + 7`.

This is how the code arrives at the solution:

- An array `d` (with a size comfortably larger than twice `n`) is used to keep track of daily changes – how many people discover and forget the secret each day.
- An array `cnt` records the number of new people who know the secret on a particular day.
- A loop runs for each day from `1` to `n` and updates `d` and `cnt` accordingly. Whenever a person learns the secret, the future changes (both the sharing and forgetting) are recorded in `d`.
- Since a person can share the secret starting from `delay` days after learning it until they forget it, a nested loop calculates all the days when the person is supposed to share the secret and updates `cnt`.
- At the end of the simulation, the sum of `d` up to day `n` (inclusive) represents the total number of people who learned (and have not forgotten) the secret. This sum is taken modulo `10^9 + 7` to get the final answer.

Solution Approach

The following is an implementation walkthrough based on the provided Python solution.

1. **Initialization**:

   - An array `d` is initialized with size `m`, where `m` is an arbitrary number greater than twice `n` to ensure enough space for the entire timeline of events (people discovering the secret and then forgetting it).
   - An array `cnt` is also initialized with the same size `m`, to track the number of new people who know the secret each day. The first person knows the secret on day `1`, so `cnt[1]` is set to `1`.
2. **Simulation Loop**:

   - A loop runs from day `1` to day `n`, simulating the spread and forgetfulness of the secret over time. If `cnt[i]` is not zero (meaning people discovered the secret on that day), the process begins.
3. **Updating Future Events**:

   - Two future events are updated in the `d` array for each person that knows the secret on day `i`: a. The day they will start sharing the secret (`i + delay`). b. The day they will forget the secret (`i + forget`), on which the count will decrease by `cnt[i]`.
4. **Sharing the Secret**:

   - A nested loop calculates all the days between the day they can start sharing (`i + delay`) and the day before they forget the secret (`i + forget`), incrementally updating `cnt[nxt]` with `cnt[i]`. This loop essentially schedules the new people who will learn the secret in the future.
5. **Modular Arithmetic**:

   - A `mod` variable is defined as `10**9 + 7`. The solution ensures that all operations involving the count use this modulus to prevent integer overflow and adhere to the problem's constraints.
6. **Final Count**:

   - At the end of the simulation, the sum of `d` array up to `n + 1` gives the total number of people that know the secret by day `n`. This sum is then taken modulo `10^9 + 7` to obtain the final result.

Notice that the solution utilizes the "prefix sum" array pattern through array `d`. By marking the start and end of an event with incremental and decremental values in `d`, it becomes possible to compute cumulative effects with only a single summation at the end. This avoids the need for multiple loops or updates within the main simulation, thereby optimizing performance.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the parameters `n` = 5 (days), `delay` = 1, and `forget` = 3. We want to find how many people will know the secret at the end of the 5 days.

1. **Initialization**:

   - We initialize the array `d` with size `m = 2*n + 1` to ensure there's more than enough space.
   - The array `cnt` is also initialized similarly, and `cnt[1]` is set to `1` because the first person knows the secret on day `1`.
2. **Simulation Start**:

   - We start iterating over the days, from day `1` to day `n` (5).

Day 1: - `cnt[1]` = 1, which means one person knows the secret on the first day. - They start sharing the secret after `delay` days, so `d[1 + delay] = d[2]` is incremented by 1 (the number of people who learned the secret on day 1). - They will forget the secret after `forget` days, so `d[1 + forget] = d[4]` is decremented by 1.

Day 2: - The person who learned the secret on day 1 starts sharing it today. - `cnt[2]` becomes `1` because `d[2]` was previously updated. - They will share it for the next couple of days, so we will have again `d[2 + delay] = d[3]` incremented by `1` and `d[2 + forget] = d[5]` decremented by 1.

Day 3: - We have one more person sharing the secret (from the sharing on day 2). - `cnt[3]` is incremented by `cnt[2]` which is 1, so `cnt[3]` is now `1`. - Similar to before, `d[3 + delay] = d[4]` is incremented by `cnt[3]` which is 1, and `d[3 + forget] = d[6]` decremented by 1.

Day 4: - Two people are sharing the secret (the original person and one more from the sharing on day 2), but the first person will forget the secret today. - `cnt[4]` is equal to `d[4]`, which is the sum of increments and decrements up to this day: `cnt[4] = d[2] + d[4]` = 1 + 1 - 1 = 1. - Again, these people will cause two increases in `d[4 + delay] = d[5]` by their count (which is 1), and `d[4 + forget] = d[7]` is decremented by 1. But remember, the first person will not contribute to future sharings since they have forgotten.

Day 5: - Now we have people who learned the secret from day 3 sharing it. - `cnt[5]` is derived from `d[5]`, which is incremented by the count from day 4 (only one person is effectively sharing since the first person forgot). - No more future updates to `d` are necessary since we have reached our last day `n`.

3. **Final Counting**:
   - Cumulate `cnt` up to day `n + 1` to determine the number of people who currently know the secret. For this example, the total is `1 + 1 + 1 + 1`, since we didn't reach day 7 when the additional decrement would happen.
   - The sum is 4, which is the total count of people who know the secret after 5 days.

The result is that 4 people know the secret at the end of day 5, respecting the `delay` and `forget` constraints. This example helps to illustrate the array updating and prefix sum technique used to solve the problem efficiently.

Python Solution

```python
class Solution:
    def peopleAwareOfSecret(self, n: int, delay: int, forget: int) -> int:
        # Initialize a large enough array to handle the propagation of information.
        max_days = (n << 1) + 10  # Bitwise left shift 'n' by 1 to double it, and add 10 for buffer
        secret_knowledge_counts = [0] * max_days  # Array to track the number of people who know the secret
        current_day_counts = [0] * max_days  # Array to track the number of people who will share the secret per day

        # On day 1, one person knows the secret.
        current_day_counts[1] = 1

        # Loop over each day up to 'n' to simulate the sharing and forgetting process.
        for day in range(1, n + 1):
            if current_day_counts[day]:  # If there are people to share the secret on this day
                # Increment the count of people who will know the secret because of sharing.
                secret_knowledge_counts[day] += current_day_counts[day]
                # Decrease the count of people who will remember the secret after they forget.
                secret_knowledge_counts[day + forget] -= current_day_counts[day]

                # Day when the current people are allowed to start sharing the secret.
                next_share_day = day + delay

                # Loop until the forget day to increment the count of people who will share the secret.
                while next_share_day < day + forget:
                    current_day_counts[next_share_day] += current_day_counts[day]
                    next_share_day += 1

        # Modulo value for the final calculation as per the problem statement, to avoid large integers.
        modulo = 10 ** 9 + 7

        # Sum all the people who know the secret up to day 'n' and return the result modulo `modulo`.
        return sum(secret_knowledge_counts[: n + 1]) % modulo

```

Java Solution

```java
class Solution {
    // Constant for modulo operation to ensure the numbers do not get too large.
    private static final int MOD = (int) 1e9 + 7;

    public int peopleAwareOfSecret(int n, int delay, int forget) {
        // A buffer is added to manage the maximum days needed.
        int bufferLength = (n << 1) + 10;

        // Daily increase tracker.
        long[] dailyIncrease = new long[bufferLength];

        // People count tracker for each day.
        long[] peopleCount = new long[bufferLength];

        // Initially, on day 1, one person knows the secret.
        peopleCount[1] = 1;

        // Loop through each day.
        for (int i = 1; i <= n; ++i) {
            // If peopleCount[i] is positive, proceed to share the secret.
            if (peopleCount[i] > 0) {
                // Add to the dailyIncrease.
                dailyIncrease[i] = (dailyIncrease[i] + peopleCount[i]) % MOD;

                // Subtract from the dailyIncrease after the forgetting period.
                dailyIncrease[i + forget] = (dailyIncrease[i + forget] - peopleCount[i] + MOD) % MOD;

                // Compute the next day when sharing starts, and continue until the forgetting day.
                int nextShareDay = i + delay;
                while (nextShareDay < i + forget) {
                    peopleCount[nextShareDay] = (peopleCount[nextShareDay] + peopleCount[i]) % MOD;
                    ++nextShareDay;
                }
            }
        }

        // Calculate the final answer by summing dailyIncrease for each day.
        long answer = 0;
        for (int i = 1; i <= n; ++i) {
            answer = (answer + dailyIncrease[i]) % MOD;
        }

        // Return the final number of people aware of the secret as an integer.
        return (int) answer;
    }
}

```

C++ Solution

```cpp
using ll = long long;
const int MOD = 1e9 + 7;

class Solution {
public:
    // This function calculates the number of people aware of a secret after 'n' days,
    // with a certain delay before they can share the secret and a forget time.
    int peopleAwareOfSecret(int n, int delay, int forget) {
        // Calculating array size to be sufficiently large
        int arraySize = (n << 1) + 10;

        vector<ll> incrementArray(arraySize, 0); // Array to handle increments of people knowing the secret
        vector<ll> peopleCountArray(arraySize, 0); // People knowing the secret by day i

        // Initially, one person knows the secret
        peopleCountArray[1] = 1;

        // Process each day
        for (int day = 1; day <= n; ++day) {
            // Skip days when no new people are aware of the secret
            if (!peopleCountArray[day]) continue;

            // Schedule an increase in the count for the current day
            incrementArray[day] = (incrementArray[day] + peopleCountArray[day]) % MOD;
            // Schedule a decrease (forgetting) in the count after the forgetting period
            incrementArray[day + forget] = (incrementArray[day + forget] - peopleCountArray[day] + MOD) % MOD;

            // Spread the secret to new people after the delay, every day until they forget
            int shareDay = day + delay;
            while (shareDay < day + forget) {
                peopleCountArray[shareDay] = (peopleCountArray[shareDay] + peopleCountArray[day]) % MOD;
                ++shareDay;
            }
        }

        // Calculate the final answer, the total number of people aware of the secret after 'n' days
        int totalAwarePeople = 0;
        for (int day = 1; day <= n; ++day) {
            totalAwarePeople = (totalAwarePeople + incrementArray[day]) % MOD;
        }
        return totalAwarePeople;
    }
};

```

Typescript Solution

```typescript
function peopleAwareOfSecret(n: number, delay: number, forget: number): number {
    // Initialize an array to track the number of people who know the secret each day.
    // Using BigInt for large numbers as the result can be quite large.
    let peopleAware = new Array(n + 1).fill(0n);

    // The first person knows the secret on the first day.
    peopleAware[1] = 1n;

    // Iterate over each day to find out how many people are aware of the secret
    for (let day = 2; day <= n; day++) {
        // Variable to store the number of people that start sharing the secret.
        let peopleSharing = 0n;

        // Calculate the number of people that will share the secret today looking back at
        // the window of days where people are able to share the secret,
        // i.e., from `day - forget + 1` to `day - delay`
        for (let shareDay = day - forget + 1; shareDay <= day - delay; shareDay++) {
            if (shareDay > 0) {
                peopleSharing += peopleAware[shareDay];
            }
        }

        // Update the current number of aware people.
        peopleAware[day] = peopleSharing;
    }

    // Variable to aggregate the total number of people who remember the secret
    // on the last day subtracting those who have forgotten
    let finalTotal = 0n;

    // Start reviewing from the last day subtracting the forget period
    for (let countDay = n - forget + 1; countDay <= n; countDay++) {
        if (countDay > 0) {
            finalTotal += peopleAware[countDay];
        }
    }

    // Return the number of people who are aware of the secret on the last day modulo 10^9 + 7, converting BigInt to number
    return Number(finalTotal % BigInt(10 ** 9 + 7));
}

```

Time and Space Complexity

The given Python function `peopleAwareOfSecret` computes the number of people aware of a secret on the `n`-th day, under certain conditions of delay before sharing and forgetting the secret. The function implements a form of dynamic programming using arrays to simulate the process over `n` days.

Time Complexity:

- The function has a primary loop that iterates over the range `1` to `n + 1`, which gives us an `O(n)` component.
- Inside this loop, there's a secondary while loop, for sharing the secret from the `i + delay`-th to the `i + forget - 1`-th day. In the worst case scenario, this while loop executes `(forget - delay)` times. Therefore, its contribution is `O(forget - delay)`.
- Since both loops are nested and the while loop runs for every value of `i`, we might initially consider the time complexity to be `O(n * (forget - delay))`.

However, the variable `nxt` is being incremented by 1 on each iteration without being reset for every `i`, and when reaching `i + forget`, the loop exits. Therefore, every element in the range `1` to `n` can only contribute to at most `(forget - delay)` increments over the entire function execution. Thus, the while-loop does not lead to a full cartesian product across `n` and `(forget - delay)`.

The actual time complexity is thus `O(n + forget)`.

Space Complexity:

- Two arrays `d` and `cnt` of maximum size `m` are used, where `m = (n << 1) + 10` - sort of a safe upper bound to ensure the array can handle the indices that the algorithm will access. These arrays are the main contributors to space complexity.
- Thus, the space complexity of the function is `O(m)`, which simplifies to `O(n)` since `m` is just a linear scaling of `n`.

To summarize:

- **Time Complexity:** `O(n + forget)`
- **Space Complexity:** `O(n)`