1482. Minimum Number of Days to Make m Bouquets

Problem Description

In this problem, we have an integer array `bloomDay` where each element represents the day on which a particular flower will bloom. We are also given two integers `m` (the number of bouquets we want to create) and `k` (the number of adjacent flowers needed to create one bouquet). The goal is to find out the minimum number of days we need to wait until we can make exactly `m` bouquets using `k` adjacent flowers for each bouquet.

The garden is comprised of `n` flowers, and the `i`-th flower will bloom on `bloomDay[i]`. We can use each flower in exactly one bouquet once it has bloomed.

If it is impossible to make `m` bouquets with the given constraints, the function should return `-1`.

Example: Given `bloomDay = [1, 10, 3, 10, 2]`, `m = 3`, `k = 1`, the output is `3` because on the third day, three flowers are bloomed which is enough to make 3 bouquets as 'k' is `1`.

Intuition

The solution to this problem involves a binary search for the minimum number of days required to make `m` bouquets. Binary search is chosen because we are dealing with an optimization problem where we want to minimize the number of days we have to wait.

Here's the intuition for why binary search can be applied:

1. We know the possible range of days is between the day the earliest flower blooms (minimum value in `bloomDay`) and the day the last flower blooms (maximum value in `bloomDay`).
2. If we can make `m` bouquets on a certain day 'x', it means we can also do it on any day after 'x' (as more flowers would be bloomed).
3. Inversely, if we cannot make `m` bouquets on a certain day 'y', we will not be able to do it on any day before 'y'.

With binary search, we can efficiently narrow down the range to find the minimum number of days required. The `check` function inside the Solution class is used to verify if a certain number of days `day` is sufficient to make `m` bouquets by scanning through `bloomDay` and counting flowers bloomed. We increment the count `cur` for consecutive bloomed flowers and when `cur` equals `k`, we increment the number of bouquets `cnt`. If `cnt` meets or exceeds `m`, the function returns `True`; otherwise, it returns `False`.

The main function then uses this `check` function to adjust the binary search boundaries (`left` and `right`) until the minimum day is found, where `left` will eventually hold the minimum day required to make `m` bouquets.

Solution Approach

The implementation of the solution makes use of a binary search algorithm and a counter pattern to verify the conditions. Below is a step-by-step walk-through of how the algorithm works:

1. We begin by checking if it is even possible to make `m` bouquets. Since each bouquet requires `k` flowers, if `m * k` is greater than the total number of flowers available (`len(bloomDay)`), it is impossible to make the required number of bouquets and we immediately return `-1`.
2. The `check` function is defined to determine if `m` bouquets can be made by a given day, say `day`. Here's how it works:
   - It initializes two counters: `cnt` (the number of bouquets that can be made by the day `day`) and `cur` (the number of adjacent bloomed flowers we are currently considering for a bouquet).
   - The function goes through each flower's bloom day in `bloomDay`.
   - For each flower, if it is bloomed by the day (`bd <= day`), we increase `cur` by 1, signifying that this flower can be part of a current bouquet.
   - If `cur` becomes equal to `k`, then we have found `k` adjacent flowers, and therefore we increment `cnt` by 1 (signifying one complete bouquet) and reset `cur` to 0 to start counting for the next bouquet.
   - If, by the end of the array, `cnt` is greater than or equal to `m`, it means we can make the required number of bouquets by the day `day`, and the function returns `True`. Otherwise, it returns `False`.
3. After defining the `check` function, binary search is employed to find the minimum number of days required to make `m` bouquets. Here's how:
   - Set `left` to the minimum value in `bloomDay` (the earliest possible day a flower can bloom) and `right` to the maximum value in `bloomDay` (the latest possible day a flower can bloom).
   - While `left` is less than `right`, we calculate the midpoint `mid` using the expression `(left + right) >> 1` which is equivalent to `(left + right) // 2`.
   - Utilize the `check` function on `mid`. If it returns `True`, it indicates that we can potentially make the bouquets even earlier, so we set `right` to `mid`. Otherwise, if `check` returns `False`, it means we need more time, so we update `left` to `mid + 1`.
   - This process continues to narrow down the range until `left` equals `right`, which would be the minimum number of days required to make the bouquets.

With each iteration, we effectively halve the search space, leading to a time complexity of $O(\log(\text{max}(\text{bloomDay}) - \text{min}(\text{bloomDay})) * n)$, where n is the time complexity of the `check` function since it scans through the entire `bloomDay` array.

Example Walkthrough

Let's go through an example to illustrate the solution approach with `bloomDay = [7, 12, 9, 5, 4]`, `m = 2` bouquets, and `k = 2` adjacent flowers needed for each bouquet. We want to find out the minimum number of days we need to wait to make exactly 2 bouquets with 2 flowers each.

Step 1: Preliminary Check

First, we verify whether it is possible to make `m` bouquets from the available number of flowers. We have `m * k = 2 * 2 = 4` flowers needed and there are 5 flowers in total (`len(bloomDay) >= 4`). Since we have more flowers than needed, it's possible to proceed.

Step 2: Setting Up Binary Search

Now, we prepare for binary search. The earliest a flower can bloom is day `4`, and the latest is day `12`. So we set `left` to `4` and `right` to `12`.

Step 3: Binary Search and the `check` Function

We will use a binary search along with the `check` function to narrow down the days.

Day 8 (First Iteration):

- Midpoint: `mid = (left + right) // 2 = (4 + 12) // 2 = 8`
- We use the `check` function to see if we can make 2 bouquets by day 8.
- Function `check bloomDay = [7, 12, 9, 5, 4], day = 8`
  - Starting from the first flower, we loop through `bloomDay`:
    - Flower 1 (`bloomDay[0]`): Blooms by day 8? Yes. `cur` is now `1`.
    - Flower 2 (`bloomDay[1]`): Blooms by day 8? No.
    - Flower 3 (`bloomDay[2]`): Blooms by day 8? No.
    - Flower 4 (`bloomDay[3]`): Blooms by day 8? Yes. `cur` is now `1` (reset because the previous flower wasn't bloomed by day 8).
    - Flower 5 (`bloomDay[4]`): Blooms by day 8? Yes. `cur` is now `2`.
  - Now `cur` equals `k`, we have a complete bouquet. So `cnt` is now `1` and `cur` resets to `0`.
  - We have finished checking all flowers and found `cnt = 1`, which is less than `m = 2`. Therefore, we cannot make enough bouquets by day 8.
- Since `check` returned `False`, we cannot make 2 bouquets by day 8, so we move our `left` boundary up. `left` is now `9`.

Day 10 (Second Iteration):

- New midpoint: `mid = (left + right) // 2 = (9 + 12) // 2 = 10`
- Function `check` with `day = 10`:
  - Loop through `bloomDay`:
    - Flower 1 blooms by day 10? Yes. `cur` is `1`.
    - Flower 2 blooms by day 10? No.
    - Flower 3 blooms by day 10? Yes. `cur` is `1` (reset because the previous flower wasn't bloomed by day 10).
    - Flower 4 blooms by day 10? Yes. `cur` is now `2` (we have the second bouquet).
    - We increment `cnt` to `2`, since we have another complete bouquet.
    - We can make the required number of bouquets by day 10, so `check` returns `True`.
- Since `check` returned `True`, we can potentially make the bouquets even earlier, so we move our `right` boundary down. `right` is now `10`.

Since `left` is now equal to `right` which is `10`, we have found the minimum number of days required to make `m` bouquets which is `10` days.

Python Solution

```python
from typing import List

class Solution:
    def minDays(self, bloom_day: List[int], m: int, k: int) -> int:
        # Early return if impossible to make m bouquets, return -1
        if m * k > len(bloom_day):
            return -1

        # Helper function to check if it's possible to make m bouquets in 'day'
        # by checking the array using a sliding window technique
        def can_make_bouquets(day: int) -> bool:
            bouquets = 0  # Number of bouquets that can be made
            flowers_in_row = 0  # Number of adjacent flowers blooming in a row

            # Loop through each bloom day
            for bloom in bloom_day:
                # If the flower will bloom by or before the chosen day, it will bloom
                if bloom <= day:
                    flowers_in_row += 1
                    # If we have enough flowers in a row, we can make a bouquet
                    if flowers_in_row == k:
                        bouquets += 1
                        flowers_in_row = 0  # Reset for the next potential bouquet
                else:
                    flowers_in_row = 0  # Reset counter if the flower won't bloom

                # Check if we can make at least m bouquets
                if bouquets >= m:
                    return True
            return False

        # Use binary search to find the minimum day to make all m bouquets
        # Start by setting the search space between the earliest and the latest bloom day
        left, right = min(bloom_day), max(bloom_day)

        # Continue searching while the search space is valid
        while left < right:
            # Find the mid-point of the search space
            mid = (left + right) // 2
            # Check if we can make the required number of bouquets by this day
            if can_make_bouquets(mid):
                # If yes, try to find a smaller day
                right = mid
            else:
                # If not, discard the left half of the search space
                left = mid + 1

        # When the search space is reduced to a single element, it's the minimum day
        return left
```

Java Solution

```java
class Solution {

    // Determines the minimum number of days required to get m bouquets of k consecutive flowers
    public int minDays(int[] bloomDay, int m, int k) {
        // If there are not enough flowers to make the required bouquets, return -1
        if (m * k > bloomDay.length) {
            return -1;
        }

        // Initialize the minimum and maximum days from the bloomDay array
        int minDays = Integer.MAX_VALUE, maxDays = Integer.MIN_VALUE;
        for (int day : bloomDay) {
            minDays = Math.min(minDays, day);
            maxDays = Math.max(maxDays, day);
        }

        // Use binary search to find the minimum day
        int left = minDays, right = maxDays;
        while (left < right) {
            int mid = left + (right - left) / 2;
            // Check if it's possible to make the bouquets by this day
            if (canMakeBouquets(bloomDay, m, k, mid)) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }

        // The left boundary of binary search will be the answer
        return left;
    }

    // Helper method to check if it's possible to make the required number of bouquets by day 'currentDay'
    private boolean canMakeBouquets(int[] bloomDay, int numBouquets, int bouquetSize, int currentDay) {
        int bouquetsMade = 0, flowersInBouquet = 0;
        for (int day : bloomDay) {
            // If the flower is bloomed by 'currentDay', include it in the current bouquet
            if (day <= currentDay) {
                flowersInBouquet++;
                // If we've gathered enough flowers for a bouquet, increase the count and reset
                if (flowersInBouquet >= bouquetSize) {
                    bouquetsMade++;
                    flowersInBouquet = 0;
                }
            } else {
                // If a flower is not bloomed, reset the count for the current bouquet
                flowersInBouquet = 0;
            }
        }
        // Check if we have made at least 'numBouquets' bouquets
        return bouquetsMade >= numBouquets;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int minDaysIncreased(int k, vector<int>& bloomDay, int bouquetCount, int flowersPerBouquet) {
        // If we cannot make the required number of bouquets, return -1
        if (bouquetCount * flowersPerBouquet > bloomDay.size()) {
            return -1;
        }

        // Get initial search range for binary search
        int minDay = *min_element(bloomDay.begin(), bloomDay.end());
        int maxDay = *max_element(bloomDay.begin(), bloomDay.end());

        int left = minDay, right = maxDay;
        while (left < right) {
            // Find the mid value of our search range
            int mid = left + (right - left) / 2;

            // Check if it is possible to make the required bouquets by mid day
            if (canMakeBouquets(bloomDay, bouquetCount, flowersPerBouquet, mid)) {
                // If yes, we can try to find a smaller day in the left half
                right = mid;
            } else {
                // If not, search in the right half
                left = mid + 1;
            }
        }

        // At this point, left is the minimum day on which we can make the bouquets
        return left;
    }

    // Helper function to verify if we can make the required number of bouquets by a given day
    bool canMakeBouquets(vector<int>& bloomDay, int bouquetCount, int flowersPerBouquet, int day) {
        int bouquetsCreated = 0, flowersCollected = 0;
        for (int flowerBloomDay : bloomDay) {
            // Collect flower if bloomed by 'day' or reset the current bouquet progress
            flowersCollected = (bloom <= day) ? (flowersCollected + 1) : 0;

            // If we have enough flowers for a bouquet, increment the count and reset flowersCollected
            if (flowersCollected >= flowersPerBouquet) {
                bouquetsCreated++;
                flowersCollected = 0;
            }
        }

        // Check if we have made at least the required number of bouquets
        return bouquetsCreated >= bouquetCount;
    }
};
```

Typescript Solution

```typescript
// Array of bloom days for individual flowers, number of bouquets to make, and number of flowers per bouquet
function minDays(bloomDay: number[], bouquetCount: number, flowersPerBouquet: number): number {
    // Cannot make the required number of bouquets with the available flowers
    if (bouquetCount * flowersPerBouquet > bloomDay.length) {
        return -1;
    }

    // Determine the lowest and highest bloom day to establish the binary search range
    let minDay = Math.min(...bloomDay);
    let maxDay = Math.max(...bloomDay);

    // Initialize binary search bounds
    let left = minDay, right = maxDay;
    while (left < right) {
        // Find the midpoint in the current search range
        const mid = Math.floor((left + right) / 2);

        // Check if we can make the required bouquets by this midpoint day
        if (canMakeBouquets(bloomDay, bouquetCount, flowersPerBouquet, mid)) {
            // Possible to make bouquets, try finding an earlier day in the left half
            right = mid;
        } else {
            // Can't make bouquets, seek a later day in the right half
            left = mid + 1;
        }
    }

    // 'left' is the earliest day we can create the required bouquets
    return left;
}

// Helper function to verify if the required number of bouquets can be made by a specific day
function canMakeBouquets(bloomDay: number[], bouquetCount: number, flowersPerBouquet: number, day: number): boolean {
    let bouquetsCreated = 0, flowersCollected = 0; // Initialize counters
    for (let flowerBloomDay of bloomDay) {
        // Count flowers that have bloomed and collected enough for a bouquet
        flowersCollected = (flowerBloomDay <= day) ? flowersCollected + 1 : 0;

        // Iterate through the array of bloom days
        for (let flowerBloomDay of bloomDay) {
            // If flower has bloomed by the given day, add to current bouquet; otherwise, reset count
            flowersCollected = (bloom <= day) ? flowersCollected + 1: 0;

            // Once enough flowers are collected for a bouquet, increment count and reset for next bouquet
            if (flowersCollected >= flowersPerBouquet) {
                bouquetsMade++;
                flowersCollected = 0; // Start collecting for the next bouquet
            }
        }

        // Return true if we can make at least the required number of bouquets
        return bouquetsMade >= bouquetCount;
    }
}
```

Time and Space Complexity

The code presented implements a binary search algorithm to find the minimum day on which we can obtain `m` bouquets of flowers, each with `k` blooms.

Time Complexity

The primary operation in this algorithm is the binary search, which operates between the range of the minimum and maximum bloom day. This operation takes $O(\log(\text{max}(\text{bloomDay}) - \text{min}(\text{bloomDay})))$. Inside each iteration of the binary search, the `check` function is called, which iterates over all the bloom days to determine if it's possible to make `m` bouquets. The iteration over bloom days in the `check` function has a time complexity of $O(n)$, where n is the number of elements in `bloomDay`.

Therefore, the overall time complexity of the algorithm is $O(n * \log(\text{max}(\text{bloomDay}) - \text{min}(\text{bloomDay})))$.

Space Complexity

The space complexity of this algorithm is determined primarily by the variables used for keeping track of the binary search bounds and the count variables within the `check` function. Since the algorithm only uses a constant amount of extra space, regardless of the input size, the space complexity is $O(1)$.