605. Can Place Flowers

Easy Greedy Array

Problem Description

In this problem, you are given an array that represents a flowerbed, where each element in the array can either be `0` or `1`. An element with a value of `0` implies that the corresponding plot in the flowerbed is empty, while an element with a value of `1` suggests that there is a flower already planted in that plot. The challenge is to plant new flowers (represented by `n`) in the empty plots under the condition that no two flowers can be adjacent to each other. If it's possible to plant `n` new flowers following this rule, you must return `true`, otherwise, you return `false`.

Intuition

The key to solving this problem is understanding that you can plant a flower in the current empty plot (`i`) only if both the preceding (`i - 1`) and following (`i + 1`) plots are also empty. This check ensures that no adjacent flowers rule is not violated.

Knowing this, we iterate through the flowerbed, and when we find an empty plot with empty adjacent plots, we plant a flower there by setting the current plot to `1` and decrementing `n`. To simplify the edge cases for the first and last plots in the flowerbed (which only have one neighbor), we can add a `0` at the start and end of the flowerbed array.

The process we follow is greedy because we plant a flower whenever we have a chance, which works since planting a flower sooner will never prevent us from planting another one later. If, by the end of this process, `n` is less than or equal to `0`, it means we have successfully planted all the required flowers without breaking the rule and we return `true`. If not, it's impossible to plant all `n` flowers with the given constraints, so we return `false`.

Solution Approach

The solution utilizes a simple algorithm with a greedy approach. To implement this solution efficiently, we modify the original array to avoid checking for edge cases separately. We add a `0` at the beginning and at the end of the `flowerbed` array. This allows us to treat the first and last plots just like any other plot in the flowerbed without risking an out-of-bounds error when checking their neighbors.

After this initial preprocessing, we iterate through the modified flowerbed array starting from index `1` and ending one element before the last. At each plot (now index `i`), we check the value of the current plot as well as its immediate neighbors. We calculate the sum of the current plot and its neighbors: `sum(flowerbed[i - 1 : i + 2])`. If this sum is `0`, it means all three consecutive plots (`i - 1`, `i`, and `i + 1`) are empty. In this case, we can safely plant a flower at the current position (`i`) by setting `flowerbed[i]` to `1` and decrementing `n` by `1` (representing planting one flower).

This loop continues till all the plots have been checked. If by the conclusion of the loop, `n` is less or equal to `0`, it signifies that we have managed to plant all required `n` flowers in accordance with the rules, hence we `return true`. If `n` is still greater than `0`, it implies that there wasn't enough space to plant all flowers while maintaining the non-adjacent condition, thus we `return false`.

This greedy approach works because by adding a flower at the earliest possible spot, we do not block any potential opportunities to add more flowers later on. Furthermore, planting a flower as soon as possible leaves us with the most options moving forward. This strategy ensures that our solution will find a valid configuration if one exists.

Example Walkthrough

Let's walk through the solution approach with a small example. Suppose we are given the following flowerbed array and `n`:

```
flowerbed = [1, 0, 0, 0, 1], n = 1
```

Following the solution approach:

1. **Preprocessing**: First, we add a `0` to the start and end of the flowerbed. Now it becomes `[0, 1, 0, 0, 0, 1, 0]`.
2. **Iterating through the flowerbed**:

- We start iteration at index `1` (the second `0`) and end at index `5` (the second last `0`).
- At index `1`, the sum of the plot and its neighbors is `0 + 1 + 0 = 1`, which is not `0`. So we don't plant here.
- At index `2`, the sum is `1 + 0 + 0 = 1`, again not `0`. So we don't plant here.
- At index `3`, the sum is `0 + 0 + 0 = 0`, which is `0`. Here, all three plots are empty. We can plant a flower here by setting flowerbed[3] to `1`. The array becomes `[0, 1, 0, 1, 0, 1, 0]`, and we decrement `n` by `1`. Now, `n = 0`.

3. **Conclusion**: We continue the iteration, but since `n` is `0`, it's clear we've managed to plant the required number of flowers without breaking the rule. There's no need to check index `4` and `5`, as we already planted all required flowers.
4. **Final Check**: Check `n`. Since `n` is now `0`, we return `true`, because it was possible to plant `1` flower following the rules.

Hence, the function would return `true` indicating that we successfully planted the required number of flowers without any two being adjacent.

Solution Implementation

Python

```python
class Solution:
    def canPlaceFlowers(self, flowerbed: List[int], n: int) -> bool:
        # Add empty plots at the start and at the end of the flowerbed to simplify edge case handling
        flowerbed = [0] + flowerbed + [0]

        # Iterate over each plot in the flowerbed starting from the first actual plot to the last
        for i in range(1, len(flowerbed) - 1):
            # Check if the current plot and its adjacent plots are empty
            if sum(flowerbed[i - 1: i + 2]) == 0:
                # Plant a flower in the current plot
                flowerbed[i] = 1
                # Decrement the count of flowers we need to plant
                n -= 1
                # If we have planted all required flowers, we can end early
                if n == 0:
                    return True

        # After checking all plots, decide if we were able to plant all flowers
        return n <= 0
```

Java

```java
class Solution {

    /**
     * Determines if n flowers can be planted in the flowerbed without violating the no-adjacent-flowers rule.
     * @param flowerbed An array representing the flowerbed where 0 is an empty spot and 1 is a spot with a flower.
     * @param n The number of flowers we want to plant.
     * @return True if we can plant n flowers, otherwise false.
     */
    public boolean canPlaceFlowers(int[] flowerbed, int n) {
        // Get the length of the flowerbed array.
        int length = flowerbed.length;

        // Iterate over all spots in the flowerbed.
        for (int i = 0; i < length; ++i) {
            // Check the spot to the left, it's 0 if we're at the first spot.
            int left = i == 0 ? 0 : flowerbed[i - 1];
            // Check the spot to the right, it's 0 if we're at the last spot.
            int right = i == length - 1 ? 0 : flowerbed[i + 1];

            // If the current, left, and right spots are all empty (i.e., 0),
            // then a flower can be planted at the current position.
            if (left + flowerbed[i] + right == 0) {
                // Plant the flower at the current position.
                flowerbed[i] = 1;
                // Decrease the remaining number of flowers to plant.
                --n;
            }
        }

        // If n is less than or equal to 0, then all flowers have been successfully planted.
        return n <= 0;
    }
}
```

C++

```cpp
class Solution {
public:
    bool canPlaceFlowers(vector<int>& flowerbed, int n) {
        int flowerbedSize = flowerbed.size(); // Size of the flowerbed.

        // Iterate through the flowerbed to find valid spots to plant flowers.
        for (int i = 0; i < flowerbedSize; ++i) {
            // Check the left neighbor (if i is 0, left neighbor is considered empty).
            int left = (i == 0) ? 0 : flowerbed[i - 1];
            // Check the right neighbor (if i is the last element, right neighbor is considered empty).
            int right = (i == flowerbedSize - 1) ? 0 : flowerbed[i + 1];

            // If both neighbors and current position are empty, we can plant a flower here.
            if (left + flowerbed[i] + right == 0) {
                flowerbed[i] = 1; // Plant a flower.
                --n; // Decrease the count of flowers needed to plant.
            }
        }
        // If n is less than or equal to 0, all flowers can be planted.
        return n <= 0;
    }
};
```

TypeScript

```typescript
// Function to determine if 'n' new flowers can be planted in a flowerbed without violating the no-adjacent-flowers rule.
function canPlaceFlowers(flowerbed: number[], n: number): boolean {
    // Get the length of the flowerbed array.
    const flowerbedLength = flowerbed.length;

    // Loop through each spot in the flowerbed.
    for (let i = 0; i < flowerbedLength; ++i) {
        // Determine the state of the left spot (0 if at the start, otherwise the previous spot).
        const leftNeighbor = i === 0 ? 0 : flowerbed[i - 1];
        // Determine the state of the right spot (0 if at the end, otherwise the next spot).
        const rightNeighbor = i === flowerbedLength - 1 ? 0 : flowerbed[i + 1];

        // Check if current spot and its neighbors are empty (0).
        if (leftNeighbor + flowerbed[i] + rightNeighbor === 0) {
            // Plant a flower at the current spot.
            flowerbed[i] = 1;
            // Decrease the count of flowers we need to plant.
            --n;
        }
    }
    // If we've managed to plant 'n' or more flowers, return true.
    return n <= 0;
}
```

```python
class Solution:
    def canPlaceFlowers(self, flowerbed: List[int], n: int) -> bool:
        # Add empty plots at the start and at the end of the flowerbed to simplify edge case handling
        flowerbed = [0] + flowerbed + [0]

        # Iterate over each plot in the flowerbed starting from the first actual plot to the last
        for i in range(1, len(flowerbed) - 1):
            # Check if the current plot and its adjacent plots are empty
            if sum(flowerbed[i - 1: i + 2]) == 0:
                # Plant a flower in the current plot
                flowerbed[i] = 1
                # Decrement the count of flowers we need to plant
                n -= 1
                # If we have planted all required flowers, we can end early
                if n == 0:
                    return True

        # After checking all plots, decide if we were able to plant all flowers
        return n <= 0
```

Time and Space Complexity

The time complexity of the provided code is indeed `O(n)`. This is because there is a single for-loop that iterates through the flowerbed list which contains `n` elements, and the operations inside the for-loop are performed in constant time.

For the space complexity, it would appear to be `O(1)` since we are not using any additional space that grows with the input size `n`. However, the list is extended at the beginning with two extra elements (`[0] + flowerbed + [0]`). Even though this operation doesn't depend on the size of the flowerbed, strictly speaking, the space complexity is `O(n)` because the input list itself is modified and could be considered as additional space being used relative to the original input list.