2136. Earliest Possible Day of Full Bloom

Hard Topics Array Greedy Sorting

Problem Description

In this problem, you have n flower seeds that you need to plant and grow. Each flower seed has two associated times:

- `plantTime[i]`: This is the number of full days it takes to plant the i-th seed. You can only work on planting one seed per day, but you don't have to do it on consecutive days. However, you cannot consider a seed as planted until you've spent the total number of `plantTime[i]` days on it.
- `growTime[i]`: This is the number of full days it takes for the i-th seed to grow and bloom after it has been completely planted. Once a flower blooms, at the end of its growth time, it stays bloomed forever.

The task is to arrange your planting strategy to ensure that all the seeds are blooming by the earliest possible day. You can plant the seeds in any order starting from day 0.

Intuition

To solve the problem, the main insight is that seeds with a longer growth time ("growTime") should be prioritised for planting. This is because the growth process can occur simultaneously with the planting of other seeds. In other words, if a seed takes a long time to grow, you want to start that growth period as soon as possible, so it doesn't delay the point in time when all flowers are blooming.

Here is the step-by-step intuition for the solution:

1. Pair each seed's planting time with its growth time and consider these as a combined attribute of the seed.
2. Sort the seeds in descending order of their growth time. By doing this, we prioritize planting seeds with the longest growth time.
3. Initialize two variables, t and `ans`, t will keep track of the total time spent planting seeds up to the current seed, and `ans` will represent the earliest day all seeds are blooming. Start with both at zero.
4. Iterate through the sorted list of pairs:
    - Increment t by the current seed's planting time. This represents the day by which planting of this seed is finished.
    - Calculate the potential blooming day for the current seed by adding its growth time to t.
    - Update `ans` to be the maximum of its current value and the potential blooming day for the current seed. This ensures that `ans` always represents the day by which all seeds seen so far will have bloomed.
5. After the iteration, `ans` will contain the earliest possible day where all seeds are blooming by following the optimal planting order.

The logic behind this approach is that by focusing on getting the long-growth seeds growing as early as possible, you minimize the waiting time at the end of the planting schedule for the flowers to bloom.

Solution Approach

To implement the solution, we utilize sorting and straightforward iteration to find the optimal sequence for planting the seeds.

- **Sorting**: We begin by pairing each `plantTime` with the corresponding `growTime` for each seed. Then, we sort the pairs based on `growTime` in descending order. This sorting places the seeds with the longest growth period first, which are the seeds we want to plant as soon as possible.

```
# Pseudocode for iSorting](//problems/sorting_summary)
paired_times = zip(plantTime, growTime)
sorted_times = sorted(paired_times, key=lambda x: -x[1])
```

- **Iteration**: Next, we iterate through our sorted pairs and simulate the planting and growing process.

```
# Pseudocode for iteration
t = 0 # Total days spent planting
ans = 0 # The earliest day all seeds are blooming
for pt, gt in sorted_times:
    t += pt # Increment total planting days by current seed's planting time
    ans = max(ans, t + gt) # Update ans to the max of current ans and the potential bloom day
```

    - During each step, we adjust our t value by adding the current seed's planting time. This t represents the time passed solely due to planting activities.
    - We then consider the total time until the current seed blooms, which is $t + gt$ (gt - growth time for the current seed). Since growth occurs after planting, it makes sense to sum these up.
    - The `ans` value is updated to ensure it represents the day when each seed planted so far will have bloomed. By using the `max` function, we ensure that it reflects the latest bloom amongst all seeds considered.

- **Variables**: Two key variables are used:
    - t: Tracks the cumulative days spent on planting.
    - `ans`: Tracks the earliest possible day where all seeds are blooming.
- **Return Value**: After the loop completes, `ans` represents the earliest day by which all the seeds would have bloomed. We arrived at this number by iteratively keeping track of the maximal end date for the blooming of each seed, hence considering the fact that while one seed is growing, we could be planting another.

The code demonstrates proper utilization of Python's list operations, such as zipping, sorting, and iteration. Moreover, the use of a simple max function elegantly resolves the problem of comparing bloom times and determining the overall earliest bloom day.

```
# Reference Solution Code
class Solution:
    def earliestFullBloom(self, plantTime: List[int], growTime: List[int]) -> int:
        ans = t = 0
        for pt, gt in sorted(zip(plantTime, growTime), key=lambda x: -x[1]):
            t += pt # day when planting of this seed is finished
            ans = max(ans, t + gt) # latest bloom day amongst all seeds
        return ans # the earliest possible day where all seeds are blooming
```

In the provided algorithm, each step is purposeful and designed to optimize the bloom time based on seed growth characteristics. By focusing on the growth period and organizing the planting schedule accordingly, we reach an optimal and efficient solution.

Example Walkthrough

Let's consider a small example to illustrate the solution approach:

Suppose we are given the following `plantTime` and `growTime` for $n = 3$ seeds:

- Seed 1: `plantTime[0] = 1`, `growTime[0] = 4`
- Seed 2: `plantTime[1] = 2`, `growTime[1] = 3`
- Seed 3: `plantTime[2] = 1`, `growTime[2] = 2`

Following the solution approach:

1. Pair each seed's planting time with its growth time:

```
Seed 1 -> (1, 4)
Seed 2 -> (2, 3)
Seed 3 -> (3, 2)
```

2. Sort the seeds in descending order of grow time:

```
Sorted Pairs:
  Seed 1 -> (1, 4)
  Seed 2 -> (2, 3)
  Seed 3 -> (3, 2)
```

(Note: Since our example is already in descending order, no changes are made).

3. Initialize t and `ans` to zero:

```
t = 0 (Total days spent planting)
ans = 0 (Earliest day all seeds are blooming)
```

4. Iterate through the sorted list of pairs:

- For Seed 1: $pt = 1, gt = 4$

```
t += 1 (t = 1)
ans = max(ans, t + gt) = max(0, 1 + 4) = 5
```

- For Seed 2: $pt = 2, gt = 3$

```
t += 2 (t = 3)
ans = max(ans, t + gt) = max(5, 3 + 3) = 6
```

- For Seed 3: $pt = 1, gt = 2$

```
t += 3 (t = 6)
ans = max(ans, t + gt) = max(6, 6 + 2) = 8
```

5. At the end of the iteration, `ans` is 6, which means all seeds will be blooming by day 8.

In this example, prioritizing the planting of seeds with the largest `growTime` allowed the seeds to start growing earlier. Even though seeds with shorter `plantTime` could have been planted first, since we prioritize by `growTime`, we minimize the overall waiting for flowers to bloom. Thus, following the optimal planting order, all seeds will be blooming by day 8.

Solution Implementation

Python

```
from typing import List

class Solution:
    def earliestFullBloom(self, plant_time: List[int], grow_time: List[int]) -> int:
        # Initialize time counters for current time `curr_time` and max time `max_time`.
        max_time = curr_time = 0

        # Pair plant and grow times, then sort by grow time in descending order.
        # This ensures that the plants with the longest grow time get planted first.
        for planting_time, growth_time in sorted(zip(plant_time, grow_time), key=lambda x: -x[1]):
            # Accumulate the current time by adding the planting time.
            curr_time += planting_time
            # Calculate the final full bloom time for the current plant and update `max_time`.
            # This is done by taking the sum of the current time and the grow time.
            max_time = max(max_time, curr_time + growth_time)

        # Return the maximum time it will take for all plants to fully bloom.
        return max_time
```

Java

```
class Solution {
    public int earliestFullBloom(int[] plantTime, int[] growTime) {
        int numPlants = plantTime.length; // Get the number of plants.
        Integer[] indices = new Integer[numPlants]; // Create an array to store the indices of plants.
        for (int i = 0; i < numPlants; ++i) {
            indices[i] = i; // Initialize indices with the index of each plant.
        }

        // Sort the array of indices based on the grow time in descending order,
        // The comparison function takes two indices and sorts them according to the grow time of their corresponding plants.
        Arrays.sort(indices, (i, j) -> growTime[j] - growTime[i]);

        int maxDays = 0; // This will store the maximum number of days needed for all plants to reach full bloom.
        int totalPlantTime = 0; // This will accumulate the total days spent planting thus far.

        // Iterate over the sorted array of indices.
        for (int i : indices) {
            // Add the time it takes to plant the current plant.
            totalPlantTime += plantTime[i];
            // Calculate the total days needed for this plant to fully bloom,
            // which is the sum of the days spent planting up to now and this plant's grow time.
            // Update maxDays to be the maximum of itself and the total bloom days of the current plant.
            maxDays = Math.max(maxDays, totalPlantTime + growTime[i]);
        }

        // Return the maximum number of days needed for all plants to reach full bloom.
        return maxDays;
    }
}
```

C++

```
class Solution {
public:
    int earliestFullBloom(vector<int>& plantTimes, vector<int>& growTimes) {
        // Get number of plants
        int numPlants = plantTimes.size();

        // Create an index vector to sort the plants based on growTime
        vector<int> indices(numPlants);
        iota(indices.begin(), indices.end(), 0);

        // Sort the indices based on grow times (descending order)
        // so that plants with longer grow times are planted first
        sort(indices.begin(), indices.end(), [&](int i, int j) {
            return growTimes[i] > growTimes[j];
        });

        // Initialize variables to track the current day and the earliest
        // day that all plants will be in full bloom
        int currentDay = 0;
        int earliestBloomDay = 0;

        // Iterate over the plants in the order determined by indices
        for (int idx : indices) {
            // Plant the current plant; this increments current planting day
            currentDay += plantTimes[idx];

            // Determine the earliest day all plants will be in full bloom
            // This is the maximum of current day + grow time of current plant
            // and the previous earliest bloom day
            earliestBloomDay = max(earliestBloomDay, currentDay + growTimes[idx]);
        }

        // Return the earliest day when all plants will be in full bloom
        return earliestBloomDay;
    }
};
```

TypeScript

```
// Function to determine the earliest day when all flowers can bloom
// by planting the optimal order in which to plant the flowers.
function earliestFullBloom(plantTimes: number[], growTimes: number[]): number {
    // Number of flowers
    const flowerCount = plantTimes.length;

    // Create an array of indices representing the flowers
    const flowerIndices: number[] = Array.from({ length: flowerCount }, (_, index) => index);

    // Sort the indices based on the growing time in descending order
    // so that we plant the flowers with the longest growing time first.
    flowerIndices.sort((indexA, indexB) => growTimes[indexB] - growTimes[indexA]);

    // Initialize variables:
    // 'totalTime' to track the total time elapsed,
    // 'maxBloomTime' to keep the maximum bloom time so far.
    let totalTime = 0;
    let maxBloomTime = 0;

    // Iterate over the sorted indices array
    for (const flowerIndex of flowerIndices) {
        // Increment the total time by the current flower's planting time
        totalTime += plantTimes[flowerIndex];
        // Update the maximum bloom time
        maxBloomTime = Math.max(maxBloomTime, totalTime + growTimes[flowerIndex]);
    }

    // Return the maximum bloom time
    // which is the earliest day when all flowers will bloom
    return maxBloomTime;
}
```

```
from typing import List

class Solution:
    def earliestFullBloom(self, plant_time: List[int], grow_time: List[int]) -> int:
        # Initialize time counters for current time `curr_time` and max time `max_time`.
        max_time = curr_time = 0

        # Pair plant and grow times, then sort by grow time in descending order.
        # This ensures that the plants with the longest grow time get planted first.
        for planting_time, growth_time in sorted(zip(plant_time, grow_time), key=lambda x: -x[1]):
            # Accumulate the current time by adding the planting time.
            curr_time += planting_time
            # Calculate the final full bloom time for the current plant and update `max_time`.
            # This is done by taking the sum of the current time and the grow time.
            max_time = max(max_time, curr_time + growth_time)

        # Return the maximum time it will take for all plants to fully bloom.
        return max_time
```

Time and Space Complexity

The given Python function `earliestFullBloom` calculates the earliest day on which all plants will be in full bloom. It does so by first sorting the combined planting and growing times in descending order of growing time, then iterating through them to calculate the total time required.

Time Complexity

The time complexity of the function is determined by the `sorted` function and the subsequent iteration through the sorted list.

1. Sorting the zipped list `sorted(zip(plantTime, growTime), key=lambda x: -x[1])` has a time complexity of $O(n \log n)$, where n represents the number of elements in `plantTime` or `growTime` lists.
2. The following for loop iterates through each element once, which has a time complexity of $O(n)$.

Combining these two steps, the overall time complexity is dominated by the sorting step, making it $O(n \log n)$.

Space Complexity

The space complexity of the function is determined by the additional space used by the sorted list and the variables inside the function.

1. The sorted list creates a new list of pairs from `plantTime` and `growTime`, having space complexity $O(n)$.
2. Variables `ans`, `t`, `pt`, and `gt` use constant space $O(1)$.

Hence, the total space complexity of the function is $O(n)$ for storing the sorted list.