1705. Maximum Number of Eaten Apples

Medium Greedy Array Heap (Priority Queue) Leetcode Link

Problem Description

In this problem, we have an apple tree that grows a certain number of apples every day for `n` days. Each apple has an expiration period given by the corresponding index in the `days` array. This means that if the tree grows `apples[i]` apples on the `i-th` day, these apples will rot and become inedible after `days[i]` days. It is also mentioned that on some days, the tree might not grow any apples, which is represented by `apples[i] == 0` and `days[i] == 0`.

You have a strategy where you eat at most one apple per day, and you're looking to maximize the number of apples you can consume. It's important to note that your apple-eating period can extend beyond the `n` days of apple growth, meaning you can continue eating the apples you've collected after the tree stops producing new ones.

The goal is to determine the maximum number of apples you can eat using the given `apples` and `days` arrays.

Intuition

The intuition behind the solution is to simulate the process of eating apples in a way that prioritizes eating the apples that will rot soonest. This helps in maximizing the number of apples eaten and minimizing waste. We can achieve this through the use of a min-heap data structure.

The min-heap will store pairs of values: the expiration date of the apples and the quantity of those apples. By ensuring that the heap is always sorted by the expiration date (the sooner-to-rot apples at the top), we can efficiently determine which apple to eat next.

Our approach follows these steps:

1. Process each day from the given arrays, starting from day 0 until the last day when apples can potentially be eaten (`i < n or q` is not empty).
2. For each day, if the tree grows apples on that day (`apples[i]`), add these apples along with their expiration date to the min-heap.
3. After that, clean the heap by removing any apples that have already rotted (`q[0][0] < i`).
4. If the heap is not empty, eat one apple (the one at the top of the heap since it's the next to rot). Decrease the quantity of that type of apple and increment the total count of eaten apples (`ans`).
5. If there are still apples of the same type left and they haven't expired yet, put them back into the min-heap with the updated quantity.
6. Repeat the above steps for each day, moving forward one day at a time (`i += 1`).
7. Continue even after day `n` until all the apples that we have are either eaten or rotten.

Using a heap is key here because it allows us to efficiently access the apples that will rot first (heap's root), eat them to prevent waste, and keep the remaining collection of apples well-organized for the upcoming days.

Solution Approach

The solution approach for this problem involves the use of a min-heap data structure, which in Python can be implemented using the `heapq` library. A min-heap helps us keep track of the apples with the earliest expiration date on top. Let's walk through the key steps in the implementation:

1. We start by initializing a heap `q` and setting `i = 0` and `ans = 0`. The variable `i` tracks the current day, and `ans` is used to count the number of apples eaten.

2. We enter a `while` loop that continues as long as `i < n` (we're within the tree's apple-producing period) or we have apples in our heap `q`.

3. Inside the loop, if it's a productive day (`apples[i] > 0`), we add the apples and their expiration day `(i + days[i] - 1)` as a tuple to the heap. We use `heappush` from the `heapq` library to maintain the heap invariant (smallest expiration date on top).

4. After potentially adding new apples to the heap, we remove any apples that have already rotted. We do this by looping while the heap is not empty and the expiration date of the apples at the top of the heap is less than the current day `q[0][0] < i`. If the condition is met, we use `heappop` to remove the top element from the heap.

5. If there are any apples left in the heap (i.e., there are still unrotten apples), we eat one. Since we can only eat one apple a day, we remove the top element from the heap, decrease the apple count by one, and increase the `ans` variable to reflect the eaten apple.

6. If there are remaining apples of the type we just ate (their count is now `v - 1`) and they have not expired (`t > i`), we push them back onto the heap with the updated count.

7. We increment `i` to move to the next day and repeat the process until the heap is empty or we've simulated all days within the apple growth period `n`.

This solution approach ensures we are always eating the apple that is closest to rotting, thus maximizing the number of apples we can consume before they go bad.

Remember, the use of a heap is due to its efficient ability to always give us the smallest element, which in this case is the soonest expiration date. The heap's property of log-complexity for insertion and removal operations enables us to manage our collection of apples as they grow and rot throughout the simulated days effectively.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the following arrays for `apples` and `days`:

```
apples = [1, 2, 3, 0, 0]
days = [3, 2, 1, 0, 0]
```

This tells us that on day `0` we have `1` apple that expires after `3` days, on day `1` we have `2` apples that expire after `2` days, and on day `2` we have `3` apples that expire after `1` day.

Initialize the heap `q`, the current day `i = 0`, and the total eaten apples `ans = 0`.

Starting at day `0`, let's walk through each day:

- **Day 0**: We have `1` apple that will expire after `3` days (i.e., it will rot on `day 3`). We add `(3, 1)` to the heap (the first value is the expiration day, and the second is the number of apples). Since there's only one apple and it's the only one in our heap, we eat it. Now, `ans = 1`.

- **Day 1**: We have `2` new apples expiring by day `3`. Add `(3, 2)` to the heap. Remove the rotten apples from the top of the heap if any (but today, there are none). Now we eat one of the apples that expire by day `3`. Thus, we update the heap to remain with `(3, 1)` for this type of apple, and `ans` becomes `2`.

- **Day 2**: We have `3` new apples that will expire by the end of today, day `2`. So we add `(2, 3)` to the heap. Before eating, we check for expired apples. Since it's day `2`, we remove the `(2, 3)` set as they have expired, and then we eat one apple from the remaining set `(3, 1)`, leaving none of that type. Now `ans = 3`.

- **Day 3**: We do not have any new apples. We check for expired apples, and the heaps top is `(3, 1)` which gets removed as it has expired today. Since there are no unexpired apples left, we can't eat any apples today.

- **Day 4**: No new apples, and the heap is empty. We don't have any apples to eat.

Therefore, using this strategy, the maximum number of apples we can eat is `3` (eaten on days `0`, `1`, and `2`).

Python Solution

```python
from heapq import heappush, heappop

class Solution:
    def eatenApples(self, apples: List[int], days: List[int]) -> int:
        # Get the total number of days for which we have apple data
        n = len(days)
        current_day = 0
        total_eaten = 0
        heap = [] # Priority queue to store apples and their expiration days

        # Continue until we process all days or run out of apples
        while current_day < n or heap:
            # If we have apples that ripen today and are not rotten, add them to the heap
            if current_day < n and apples[current_day] > 0:
                # Store tuples of (expiration day, number of apples)
                heappush(heap, (current_day + days[current_day] - 1, apples[current_day]))

            # Remove rotten apples from the top of the heap (if any)
            while heap and heap[0][0] < current_day:
                heappop(heap)

            # If we have any fresh apples, eat one
            if heap:
                expiration_day, apple_count = heappop(heap)
                apple_count -= 1  # Eating one apple
                total_eaten += 1  # Increase the total eaten apples count
                # If there are apples remaining and they aren't expired, put them back in the heap
                if apple_count > 0 and expiration_day > current_day:
                    heappush(heap, (expiration_day, apple_count))

            # Move to the next day
            current_day += 1

        # Return the total number of apples eaten
        return total_eaten

# Example usage:
# apples = [1, 2, 3, 5, 2]
# days = [3, 2, 1, 4, 2]
# solution = Solution()
# print(solution.eatenApples(apples, days))  # Output will be the total apples eaten based on the input arrays

```

Java Solution

```java
class Solution {
    public int eatenApples(int[] apples, int[] days) {
        // PriorityQueue to store apple batches with their expiry days as the priority.
        // It uses a comparator to ensure that the batch with the earliest expiry (smallest day) is at the top.
        PriorityQueue<int[]> appleQueue = new PriorityQueue<>(Comparator.comparingInt(a -> a[0]));

        int n = apples.length; // Number of days for which we have apple availability data.
        int totalEatenApples = 0; // Counter to keep track of total apples eaten.
        int currentDay = 0; // Current day, starting from day 0.

        // Loop through each day until we have processed all days or the queue is empty.
        while (currentDay < n || !appleQueue.isEmpty()) {
            // If apples are available on the current day, add them to the queue.
            if (currentDay < n && apples[currentDay] > 0) {
                appleQueue.offer(new int[] {currentDay + days[currentDay] - 1, apples[currentDay]});
            }

            // Remove all batches from the queue that have expired by the current day.
            while (!appleQueue.isEmpty() && appleQueue.peek()[0] < currentDay) {
                appleQueue.poll();
            }

            // If there is at least one batch of apples that hasn't expired, eat one apple.
            if (!appleQueue.isEmpty()) {
                int[] batch = appleQueue.poll(); // Get the batch with the earliest expiry date.
                totalEatenApples++; // Increment the count of eaten apples.
                batch[1]--; // Decrement the count of apples in the batch since one is eaten.

                // If there are still apples left in the batch and it hasn't expired, put it back in the queue.
                if (batch[1] > 0 && batch[0] > currentDay) {
                    appleQueue.offer(batch);
                }
            }
            // Move to the next day.
            currentDay++;
        }

        return totalEatenApples; // Return the total number of apples eaten.
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <queue>

using std::vector;
using std::priority_queue;
using std::pair;
using std::greater;

class Solution {
public:
    int eatenApples(vector<int>& apples, vector<int>& days) {
        // Priority queue to store apples with their expiration and count.
        // The pair consists of the expiration day and the number of apples.
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> appleQueue;
        int totalApplesEaten = 0;
        int currentDate = 0;
        int totalDays = days.size(); // Number of days we have apple and day information for.

        // Continue until we have processed all days or until the queue is empty.
        while (currentDate < totalDays || !appleQueue.empty()) {
            // If we are within the day range and there are apples available on this day,
            // add them to the priority queue with their expiration date.
            if (currentDate < totalDays && apples[currentDate] > 0) {
                int expirationDay = currentDate + days[currentDate] - 1;
                appleQueue.emplace(expirationDay, apples[currentDate]);
            }

            // Remove all expired apples from the queue.
            while (!appleQueue.empty() && appleQueue.top().first < currentDate) {
                appleQueue.pop(); // Discard the expired apples.
            }

            // If there are any fresh apples left, eat one and update the count.
            if (!appleQueue.empty()) {
                auto [expiration, count] = appleQueue.top();
                appleQueue.pop();
                count--; // Eat one apple.
                totalApplesEaten++; // Increment total eaten apple count.
                if (count > 0 && expiration > currentDate) {
                    // If there are apples left that haven't expired, put them back into the queue.
                    appleQueue.emplace(expiration, count);
                }
            }

            currentDate++; // Move to the next day.
        }

        return totalApplesEaten; // Return the total number of apples eaten.
    }
};

```

Typescript Solution

```typescript
// We are importing specific required types from the 'typescript-collections' library for priority queue implementation.
import { PriorityQueue } from 'typescript-collections';

interface AppleInfo {
    expirationDay: number;
    count: number;
}

// Eaten apples calculation function using vectors representing the number of apples and their days to rot.
function eatenApples(apples: number[], days: number[]): number {
    // Priority queue for apples based on their expiration, with the earliest expiration at the top.
    const appleQueue = new PriorityQueue<AppleInfo>((a, b) => a.expirationDay - b.expirationDay);
    let totalApplesEaten: number = 0;
    let currentDate: number = 0;
    // Calculate the number of days we have apple and day information.
    const totalDays: number = days.length;

    // Continue until all days are processed, or until the queue is empty.
    while (currentDate < totalDays || !appleQueue.isEmpty()) {
        // If within the days range and apples are available, add them with their expiration date.
        if (currentDate < totalDays && apples[currentDate] > 0) {
            const expirationDay = currentDate + days[currentDate] - 1;
            appleQueue.enqueue({ expirationDay, count: apples[currentDate] });
        }

        // Remove all expired apples from the queue.
        while (!appleQueue.isEmpty() && appleQueue.peek().expirationDay < currentDate) {
            appleQueue.dequeue();
        }

        // If there are fresh apples, eat one and update the count.
        if (!appleQueue.isEmpty()) {
            let appleInfo = appleQueue.dequeue();
            appleInfo.count--;
            totalApplesEaten++;

            // If there are uneaten apples which have not expired, put them back.
            if (appleInfo.count > 0 && appleInfo.expirationDay > currentDate) {
                appleQueue.enqueue(appleInfo);
            }
        }

        // Move to the next day.
        currentDate++;
    }

    // Return the total number of apples eaten.
    return totalApplesEaten;
}

```

Time and Space Complexity

The time complexity of the code is `O(n + k log k)`, where `n` is the length of the `days` array, and `k` is the total number of different apples received over the days. Each day up to day `n`, the operation either pushes (`heappush`) or pops (`heappop`) an item from the heap `q`, which takes `O(log k)` time. After day `n`, all elements in the heap `q` will be popped without any further pushes, so the complexity in that phase depends on the number of elements in the heap, which can be at most `k`. Since each element in the heap can only be pushed and popped once, the number of heap operations is limited to the number of elements `k`. Therefore, the total time spent on heap operations is `O(k log k)`. Additionally, there is a constant amount of work done for each day `i` up to day `n`, contributing to the `O(n)` factor.

The space complexity of the code is `O(k)`, as it requires storing each different apple's expiry date and count in the heap `q`. In the worst case, `k` could be as large as the number of days if we receive at least one apple on each day with different expiry dates. The heap size will never exceed the total number of apples because once an apple's expiry date has passed, it is removed from the heap.