2126. Destroying Asteroids

Medium Greedy Array Sorting Leetcode Link

Problem Description

In this problem, we're dealing with a scenario where you have a planet with a given mass and a sequence of asteroids, each with their own mass. The crux of the problem is to determine whether the planet can survive collisions with all of the asteroids, one by one, in any order of our choosing. When a collision with an asteroid occurs, if the planet's mass is at least as much as that asteroid, the asteroid gets destroyed, and the planet absorbs its mass, becoming more massive. However, if the asteroid's mass is greater, the planet would be destroyed upon impact. Our task is to decide whether we can find a sequence in which all asteroids can be destroyed by the planet without the planet being destroyed in the process.

Intuition

To solve this problem, we focus on a greedy strategy. The intuition here is quite simple: start with the smallest asteroid. If the planet can't destroy the smallest asteroid, there is no way it can destroy the larger ones. Because collisions are not sequence specific, we can tackle asteroids in ascending order of mass without impacting the overall outcome.

Following this strategy, we should first sort the asteroids by their mass in ascending order. Then, we iterate through the sorted list, at each step checking if the planet's mass is sufficient to destroy the current asteroid. If it is, we add this asteroid's mass to the planet, effectively increasing its ability to destroy subsequent, possibly larger asteroids. If, at any point, we encounter an asteroid that is too big to be destroyed (i.e., its mass is greater than the planet's current mass), we know that the sequence in which the rest of the collisions occur is irrelevant — the planet simply can't destroy all of the asteroids.

Here's the reasoning in steps:

1. Sort `asteroids` in increasing order to prioritize absorbing smaller asteroids first.
2. For each asteroid in the sorted `asteroids`:
   - If the planet's mass is less than the current asteroid's mass, return `False` as the planet will be destroyed.
   - If the planet's mass is greater or equal to the asteroid's mass, add the asteroid's mass to the planet's mass.
3. If all asteroids are successfully absorbed, return `True`.

The solution provided confirms to this approach, ensuring the planet's survival is determined optimally.

Solution Approach

The implementation of the proposed solution takes advantage of a very simple yet effective idea from the field of algorithms — specifically, a greedy strategy requires sorting to work correctly.

Here are the steps involved in implementing the solution:

1. The `asteroids` array is sorted using Python's built-in `.sort()` method, which sorts the array in-place in ascending order. Sorting is essential because it allows us to begin absorbing asteroids from smallest to largest, therefore ensuring the planet accumulates as much mass as possible before encountering larger asteroids.

2. A `for` loop is then used to iterate over the sorted `asteroids`. The `for` loop checks each asteroid's mass against the current mass of the planet.

3. Inside the loop, there's an `if` statement that checks whether the current asteroid's mass is greater than the planet's mass. If that's the case (`mass < v`), then it's not possible for the planet to survive the collision with this asteroid, and the function immediately returns `False`.

4. If the planet's mass is sufficient to destroy the asteroid (`mass >= v`), it absorbs the asteroid's mass (`mass += v`). The planet's mass is incremented by the mass of the asteroid it just destroyed, enabling us to potentially destroy larger asteroids in the subsequent iterations.

5. After the `for` loop concludes (if we haven't returned `False` for any asteroid), it means the planet was able to destroy all asteroids in ascending order of their mass and thus able to survive. Hence, the function returns `True`.

6. The overall time complexity of the solution is governed by the sorting process, which, in Python, performs at `O(n log n)` complexity, with `n` being the number of asteroids.

This problem doesn't require any additional data structures; a list to hold the input array and a single variable to track the planet's mass are sufficient. The pattern used here is a common one in algorithm design: make the locally optimal choice at each iteration with the hope of finding the global optimum — a hallmark of greedy algorithms.

Here is the relevant portion of the solution code for reference:

```python
class Solution:
    def asteroidsDestroyed(self, mass: int, asteroids: List[int]) -> bool:
        asteroids.sort()
        for v in asteroids:
            if mass < v:
                return False
            mass += v
        return True
```

Each element of the `asteroids` list is examined exactly once after the list has been sorted, and this approach guarantees that the planet will destroy the asteroids in the most optimal sequence.

Example Walkthrough

Let's use a small example to illustrate the solution approach.

Imagine a planet with a mass of 10, and a sequence of asteroids with the following masses: `[3, 9, 19, 5, 12]`. We want to determine if the planet can survive the impact of all these asteroids one by one, absorbing any that it can destroy.

First, we employ a greedy strategy and sort the asteroids by mass in ascending order: Sorted asteroids: `[3, 5, 9, 12, 19]`.

Now, let's walk through the sorted list and simulate the planet's collisions with the asteroids:

1. The first asteroid has a mass of 3, which is less than the planet's mass of 10. The planet can destroy it and absorb its mass.
   - New planet mass: `10 + 3 = 13`.
2. The next asteroid has a mass of 5, which is less than the planet's new mass of 13. The planet can destroy it as well.
   - New planet mass: `13 + 5 = 18`.
3. The third asteroid's mass is 9, and again, it is less than the planet's current mass of 18. The planet destroys and absorbs this asteroid too.
   - New planet mass: `18 + 9 = 27`.
4. The fourth asteroid has a mass of 12, which is also less than the planet's mass of 27. The planet continues to absorb the asteroid's mass.
   - New planet mass: `27 + 12 = 39`.
5. Finally, the last asteroid has a mass of 19. As the planet's mass is now 39, it can easily destroy and absorb this asteroid.
   - New planet mass: `39 + 19 = 58`.

Since the planet has successfully absorbed each asteroid in turn, growing more massive each time, it can survive all collisions in this specific sequence. Therefore, the function will return `True` - the planet survives.

Here's how the code will execute this example:

```python
class Solution:
    def asteroidsDestroyed(self, mass: int, asteroids: List[int]) -> bool:
        asteroids.sort()  # [3, 5, 9, 12, 19]
        for v in asteroids:
            if mass < v:  # This check will fail for all asteroids in this example
                return False
            mass += v  # Planet's mass incrementally grows: 13, 18, 27, 39, 58
        return True  # The planet survived all asteroids.

# Test the example
solution = Solution()
assert solution.asteroidsDestroyed(10, [3, 9, 19, 5, 12]) == True
```

This example demonstrated the greedy strategy in action. By sorting the asteroids by mass and absorbing them starting with the smallest mass, the planet was able to successfully destroy and absorb all incoming asteroids.

Python Solution

```python
from typing import List

class Solution:
    def asteroidsDestroyed(self, current_mass: int, asteroids: List[int]) -> bool:
        # Sort the list of asteroids in ascending order based on their mass.
        asteroids.sort()

        # Iterate through the sorted list of asteroids.
        for asteroid_mass in asteroids:
            # If the current mass is less than the next asteroid's mass,
            # the asteroid cannot be destroyed, hence return False.
            if current_mass < asteroid_mass:
                return False

            # Add the mass of the destroyed asteroid to the current mass.
            current_mass += asteroid_mass

        # If all asteroids have been successfully destroyed, return True.
        return True

# Example usage:
# solution = Solution()
# result = solution.asteroidsDestroyed(10, [3, 9, 19, 5, 21])
# print(result)  # This would print False, as the 19 and 21 cannot be destroyed.

```

Java Solution

```java
import java.util.Arrays; // Import Arrays class for sorting.

class Solution {
    // Method to determine if all asteroids can be destroyed.
    public boolean asteroidsDestroyed(int mass, int[] asteroids) {
        // Sort the asteroids array to process them in ascending order.
        Arrays.sort(asteroids);

        // Use long to avoid integer overflow as mass might become larger than int range.
        long currentMass = mass;

        // Iterate through the sorted asteroids.
        for (int asteroidMass : asteroids) {
            // If the current asteroid mass is larger than the current mass, destruction is not possible.
            if (currentMass < asteroidMass) {
                return false;
            }
            // Otherwise, add the asteroid's mass to the current mass.
            currentMass += asteroidMass;
        }

        // Return true if all asteroids have been successfully destroyed.
        return true;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // For std::sort

class Solution {
public:
    // Function checks if all asteroids can be destroyed given the initial mass
    bool asteroidsDestroyed(int initialMass, vector<int>& asteroids) {
        // Sort the asteroids in non-decreasing order to handle smaller asteroids first
        std::sort(asteroids.begin(), asteroids.end());

        // Use a long long type for currentMass to avoid overflow issues
        long long currentMass = initialMass;

        // Loop through the sorted asteroids
        for (int asteroidMass : asteroids) {
            // If the current asteroid is heavier than the current mass, return false
            if (currentMass < asteroidMass) {
                return false;
            }
            // Otherwise, add the mass of the destroyed asteroid to the current mass
            currentMass += asteroidMass;
        }

        // If all asteroids have been successfully destroyed, return true
        return true;
    }
};

```

Typescript Solution

```typescript
// Include typing for the asteroid array
type AsteroidArray = number[];

// Define a global variable for initial mass (assuming we want to track it globally)
let initialMass: number = 0;

// Function checks if all asteroids can be destroyed given the initial mass
function asteroidsDestroyed(mass: number, asteroids: AsteroidArray): boolean {
    // Sort the asteroids in non-decreasing order to handle smaller asteroids first
    asteroids.sort((a, b) => a - b);

    // Use a bigint type for currentMass to avoid overflow issues
    let currentMass: bigint = BigInt(mass);

    // Loop through the sorted asteroids
    for (let asteroidMass of asteroids) {
        let massBigInt = BigInt(asteroidMass);

        // If the current asteroid is heavier than the current mass, return false
        if (currentMass < massBigInt) {
            return false;
        }

        // Otherwise, add the mass of the destroyed asteroid to the current mass
        currentMass += massBigInt;
    }

    // If all asteroids have been successfully destroyed, return true
    return true;
}

```

Time and Space Complexity

The time complexity of the code is `O(n log n)` where `n` is the number of asteroids. This is because the most time-consuming operation is the sort function, which typically has `O(n log n)` complexity.

The space complexity of the code is `O(1)` assuming that the sort is done in-place (as is typical with Python's sort on lists). This means that aside from a constant amount of additional space, the code does not require extra space that scales with the input size.