1921. Eliminate Maximum Number of Monsters

Medium Greedy Array Sorting

Leetcode Link

Problem Description

In this problem, you are cast as the defender of a city which is under attack by a horde of monsters. Each monster is approaching the city from a certain distance away, and each one moves at its own constant speed. You have a weapon that can eliminate one monster at a time, but it needs one minute to charge before it can be used again. The game ends in a loss if any monster manages to reach the city, and a monster arriving at the same time as the weapon charges is also considered a loss.

You are given two arrays: `dist`, which contains the initial distances of each monster from the city, and `speed`, which contains the speeds of each monster. Your objective is to determine the maximum number of monsters you can eliminate before any one monster reaches the city, or to confirm that you can eliminate all monsters before they reach the city.

The essence of the problem is to optimize the order in which you eliminate the monsters to maximize the number you can defeat before any reach the city.

Intuition

The intuitive approach to this problem is about timing. We want to calculate the time it will take for each monster to reach the city and then prioritize the elimination of monsters based on how soon they will arrive.

Since the weapon takes one minute to charge, we can think of each minute as a round in which we can eliminate exactly one monster. To maximize the number of monsters we can eliminate, we should always choose to eliminate the one that would reach the city soonest in the next round.

Thus, we calculate the arrival time for each monster by taking their distance and dividing it by their speed. This gives us the time in minutes when the monsters would reach the city if they were not stopped.

We sort these times because it's crucial to deal with the monsters that have the smallest arrival times first. This is because a monster with a smaller arrival time poses a more immediate threat to the city.

The solution iterates over the sorted list of arrival times, simulating the passage of each minute (each iteration is a minute passing). It compares the time needed for each monster to reach the city with the elapsed time (`i`). If at any minute the arrival time of a monster is less than or equal to the current minute (`t < i`), it means a monster has reached the city and we can return the number of monsters eliminated by that minute.

If we successfully pass through the entire list without any monster reaching the city, it means we can eliminate all of them, and we return the total number of monsters (`len(times)`).

Solution Approach

The implementation of the solution follows these steps, which makes use of basic algorithmic concepts and Python's list operations:

1. Pair each monster's distance with its speed using Python's `zip` function. This pairs up each distance `d` in `dist` with the corresponding speed `s` in `speed` on a one-to-one basis, resulting in a tuple `(d, s)` for each monster.
2. Calculate the time it will take for each monster to reach the city. This is done by dividing each monster's distance by its speed, `d / s`, but since the monster is defeated if the weapon charges at the same time it arrives, we reduce the distance by 1 (`(d - 1)`). This handles scenarios where the monster would arrive exactly as the weapon charges (considered a loss). In Python, standard division produces a float, but we are interested in the integer division (`//`), which gives us the number of completed minutes before the monster arrives.
3. We then sort the calculated times. Sorting is important here because it allows us to process the monsters in the order they will arrive.
4. We iterate over the sorted times, where `i` represents the current minute/round and `t` represents the time at which the monster will reach the city.
5. During the iteration, we check if `t < i`. If this condition is true, it means a monster will reach the city before we have the opportunity to eliminate it in the current round, and we must stop the game. The number of rounds elapsed (`i`) is the maximum number of monsters we can eliminate.
6. If the loop completes without triggering the stop condition, it means all monsters can be eliminated before any reach the city. Thus, we return `len(times)`, which is the total number of monsters.

This solution takes advantage of the sorted list data structure to easily iterate through the monsters in the order we want to process them (from the soonest to arrive to the latest). It also utilizes simple arithmetic and logical comparisons to determine the outcome at each minute of the game.

Example Walkthrough

Let's assume we have the following arrays of distances `dist` and `speeds` for the monsters:

- `dist = [8, 12, 24]`
- `speed = [4, 4, 4]`

First, we need to pair each monster's distance with its speed using Python's `zip` function, resulting in the pairs:

- Pairs: `[(8, 4), (12, 4), (24, 4)]`

Next, we calculate the time it will take each monster to reach the city, being careful to subtract 1 from the distance to handle scenarios where the monster arrives exactly as the weapon charges:

- Calculations for time: `[(8 - 1) // 4, (12 - 1) // 4, (24 - 1) // 4]`
- Resulting arrival times: `[1, 2, 5]`

Now, we need to sort these times in ascending order to determine the order in which we will attempt to eliminate the monsters:

- Sorted times: `[1, 2, 5]`

Starting at minute 0, we iterate over these times, and `t` represents a monster's arrival time, while `i` represents the current minute (round):

- Round 0: `i = 0`, monster arrival times `[1, 2, 5]`, no monster is eliminated yet.
- Round 1: `i = 1`, eliminate the monster arriving at time `1`. Remaining times `[2, 5]`.
- Round 2: `i = 2`, eliminate the monster arriving at time `2`. Remaining time `[5]`.
- Round 3: `i = 3`, no action, as no monster is arriving this minute.
- Round 4: `i = 4`, no action again.
- Round 5: `i = 5`, eliminate the monster arriving at time `5`. No remaining monsters.

As we iterate through the sorted list of arrival times, we check at each step if `t < i`. However, during our game, this never occurs as we can defeat each monster in their respective round before any of them reach the city. Since we have successfully eliminated all monsters, we can safely return the total number of monsters, which, in this case, is `3`.

Thus, the city is successfully defended, and all monsters are defeated.

Python Solution

```python
from typing import List

class Solution:
    def eliminateMaximum(self, distances: List[int], speeds: List[int]) -> int:
        # Calculate the time it takes for each monster to reach the player
        arrival_times = sorted((distance - 1) // speed for distance, speed in zip(distances, speeds))

        # Iterate through the sorted arrival times
        for monster_index, arrival_time in enumerate(arrival_times):
            # If the arrival time is less than the number of monsters eliminated,
            # that means we cannot eliminate this monster before it reaches the player
            if arrival_time < monster_index:
                # Return the number of monsters eliminated before this one
                return monster_index

        # If all monsters can be eliminated, return the total number
        # This is equal to the length of the arrival times list
        return len(arrival_times)

```

Java Solution

```java
class Solution {

    public int eliminateMaximum(int[] dist, int[] speed) {
        int monsterCount = dist.length; // Number of monsters
        int[] arrivalTimes = new int[monsterCount]; // Store times when each monster will arrive

        // Calculate the arrival time for each monster, rounded down
        for (int i = 0; i < monsterCount; ++i) {
            // '-1' because we can defeat a monster at the start of the time unit before it reaches us
            arrivalTimes[i] = (dist[i] - 1) / speed[i];
        }

        // Sort the monsters by their arrival times in ascending order
        Arrays.sort(arrivalTimes);

        // Go through the sorted list to find how many monsters can be eliminated
        for (int i = 0; i < monsterCount; ++i) {
            // If a monster's arrival time is less than the time units spent, you can't eliminate it
            if (arrivalTimes[i] < i) {
                return i; // Return the number of monsters defeated before the player is caught
            }
        }

        // If all monsters' arrival times are greater than or equal to time spent, all can be defeated
        return monsterCount;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int eliminateMaximum(vector<int>& distances, vector<int>& speeds) {
        int numMonsters = distances.size(); // Number of monsters
        vector<int> arrivalTimes; // Store the times at which each monster arrives

        // Calculate the arrival time for each monster and store it
        for (int i = 0; i < numMonsters; ++i) {
            // Calculate arrival time and subtract 1 because a monster is eliminated at the beginning of the turn
            // So if a monster arrives exactly at a turn, it can be eliminated just before it attacks
            arrivalTimes.push_back((distances[i] - 1) / speeds[i]);
        }

        // Sort the arrival times in ascending order
        sort(arrivalTimes.begin(), arrivalTimes.end());

        // Iterate through the sorted arrival times
        for (int i = 0; i < numMonsters; ++i) {
            // If a monster's arrival time is less than the current time 'i' (the turn),
            // that monster cannot be eliminated before it attacks
            if (arrivalTimes[i] < i) {
                return i; // Return the number of monsters eliminated before any monster attacks
            }
        }

        // If all monsters can be eliminated one per turn before they attack, return the total number of monsters
        return numMonsters;
    }
};

```

Typescript Solution

```typescript
// Function to determine the maximum number of monsters that can be eliminated
// before any of them reaches the player, given their distances and speeds.
function eliminateMaximum(distances: number[], speeds: number[]): number {
    const monsterCount = distances.length;

    // Array to hold the time at which each monster will reach the player
    const arrivalTimes = new Array(monsterCount).fill(0);

    // Calculate the arrival time for each monster
    for (let i = 0; i < monsterCount; ++i) {
        // Subtracting 1 from distance to avoid ceiling
        // because we start counting from zero
        arrivalTimes[i] = Math.floor((distances[i] - 1) / speeds[i]);
    }

    // Sort the arrival times in ascending order to prioritize which monsters to eliminate first
    arrivalTimes.sort((a, b) => a - b);

    // Iterate over the sorted arrival times
    for (let i = 0; i < monsterCount; ++i) {
        // If a monster's arrival time is less than the time taken to eliminate monsters so far,
        // return the number of monsters that have been eliminated until this point
        if (arrivalTimes[i] < i) {
            return i;
        }
    }

    // If all monsters can be eliminated before any reach the player, return the total count
    return monsterCount;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by several factors:

1. Calculating `times` list, which includes iterating over both `dist` and `speed` lists, and performs a division as well as subtraction for each element. This operation is `O(n)`, where `n` is the number of elements in `dist` or `speed` (assuming they are of the same length).
2. Sorting the `times` list. The sorting operation is the most time-consuming one and it typically takes `O(n log n)` time using Timsort (the default sorting algorithm in Python).
3. Finally, there is a loop to check for `t < i`, which is at most `O(n)` since it iterates through the `times` list.

Combining these operations, the most time-consuming operation is the sorting, thus the total time complexity of the algorithm is `O(n log n)`.

Space Complexity

The space complexity involves:

1. Additional space for the `times` list which is `O(n)`.
2. Constant extra space for the loop iteration variable and the return value, which is `O(1)`.

Hence, the total space complexity is dominated by the `times` list, resulting in `O(n)`.