2383. Minimum Hours of Training to Win a Competition

Easy Greedy Array

Leetcode Link

Problem Description

In this competition, you're equipped with two initial resources: energy and experience, which are both positive integers. You'll be contending against `n` challengers, whose energy and experience levels are listed in two separate arrays named `energy` and `experience`, respectively. These arrays are matched in length, representing the sequence of opponents you'll face.

Your goal is to overcome each adversary sequentially. To successfully defeat an adversary, you must possess more energy and more experience than they do. When you overcome an opponent, it will cost you some of your energy (specifically, `energy[i]`), but you'll gain experience concretely, `experience[i]`. There's a catch, though; if your current levels of energy or experience are not sufficient to beat an opponent, you have the option to train.

Training can be done as much as needed before the competition starts. Each hour of training allows you to boost either your energy or your experience by one unit. The problem is to determine the minimal number of hours you need to train to guarantee victory over all `n` opponents.

Intuition

To solve this problem, we need to ensure that before each fight, our energy and experience exceed those of the current opponent. To achieve this, we may need to engage in some preemptive training.

The intuition behind the solution involves iterating over each opponent and checking if our current energy and experience is sufficient to win. For energy, if we have less or equal energy compared to the current opponent's energy, we must train until our energy is greater by at least one. Similarly, for experience, if our experience is less or equal to that of the current opponent, we train until our experience exceeds the opponent's by at least one.

After ensuring we can defeat the opponent, we then engage in the battle, which results in a decrease in energy by `energy[i]` and an increase in experience by `experience[i]`. We continue this process, battle by battle, keeping track of the total hours spent training, until we are capable of defeating all opponents.

Thus, the solution is a simple simulation that accumulates the total number of training hours necessary as we iterate over the array of opponents.

Solution Approach

The implementation provided is straightforward and does not use complex data structures or algorithms. It simply iterates through the two input arrays - `energy` and `experience` - and directly modifies the `initialEnergy` and `initialExperience` variables while keeping track of the additional training hours needed with the `ans` variable. The procedural steps can be broken down as follows:

1. Initialize `ans` to `0`. This variable will accumulate the total hours of training needed.
2. Loop through the `energy` and `experience` arrays simultaneously using Python's built-in `zip` function. This allows us to examine the energy and experience of each opponent in the sequence they are encountered.
3. For each opponent, compare `initialEnergy` with the opponent's energy (`a`):
   - If `initialEnergy` is less than or equal to `a`, calculate the difference between the opponent's energy and yours, add one to it (to ensure it's strictly greater), and add that to `ans`.
   - Then, set `initialEnergy` to the opponent's energy plus one for the same reason.
4. Perform a similar operation for experience. Compare `initialExperience` with the opponent's experience (`b`):
   - If `initialExperience` is less than or equal to `b`, calculate the difference, add one, and increment `ans` with this value.
   - Update `initialExperience` to the opponent's experience plus one.
5. After adjusting for any needed training, simulate the battle by decreasing `initialEnergy` by `a` (the opponent's energy) and increasing `initialExperience` by `b` (the opponent's experience).
6. Repeat steps 3 to 5 for each opponent until you have iterated through all elements of `energy` and `experience` arrays.
7. Once all opponents have been considered, return `ans` which now contains the minimum number of training hours required to defeat all opponents.

By using simple conditional checks and updating variables on-the-fly, this approach simulates each match's outcome while accommodating any necessary training beforehand. No extra space is required beyond the function's parameters and local variables, making the space complexity O(1). The time complexity is O(n), where n is the number of opponents, since we need to perform a constant amount of work per opponent.

Example Walkthrough

Assume our initial energy is 15 and our initial experience is 10. We face 3 opponents with the following profiles:

- Opponent 1: Energy = 5, Experience = 3
- Opponent 2: Energy = 14, Experience = 8
- Opponent 3: Energy = 10, Experience = 15

We start by comparing our energy and experience to those of Opponent 1.

1. We have 15 energy and 10 experience, which is more than Opponent 1, so no training is needed. After defeating Opponent 1, we subtract their energy from ours and add their experience to ours. Our new energy and experience are 10 (15 - 5) and 13 (10 + 3), respectively.

2. Our energy is weaker than Opponent 2 (10 < 14), so we need training. We train for 5 hours to raise our energy to 15 (10 + 5). We have 15 energy and 13 experience facing Opponent 2, so no experience training is needed. After the match, our energy is 1 (15 - 14) and our experience is 21 (13 + 8).

3. Facing Opponent 3, our energy is low. We must train for 10 hours to get to 11 energy (1 + 10). However, our experience (21) is already higher than Opponent 3's, so no training for experience is needed. After this match, we end with 1 energy (11 - 10) and 36 experience (21 + 15).

Through this process, we trained for 15 hours (5 hours for energy against Opponent 2 and 10 hours for energy against Opponent 3), ensuring victory against all opponents. This example confirms the strategy of checking each opponent's energy and experience, training as needed, and then recalculating our stats after each battle to be prepared for the next one.

Python Solution

```python
from typing import List

class Solution:
    def minNumberOfHours(
        self,
        initial_energy: int,
        initial_experience: int,
        energy_required: List[int],
        experience_gained: List[int],
    ) -> int:
        # Initialize the number of hours the hero needs to train to 0.
        total_hours_needed = 0

        # Iterate over the battles the hero needs to fight.
        for required_energy, gained_experience in zip(energy_required, experience_gained):
            # Check if the hero's current energy is less than or equal to the required energy for the battle.
            if initial_energy <= required_energy:
                # Calculate the energy shortfall and increment hours needed to at least one more than required.
                energy_shortfall = required_energy - initial_energy + 1
                total_hours_needed += energy_shortfall
                # Update the hero's energy level after training.
                initial_energy += energy_shortfall

            # Check if the hero's current experience is less than or equal to the experience of the enemy.
            if initial_experience <= gained_experience:
                # Calculate the experience shortfall and increment hours needed to at least one more than the enemy's.
                experience_shortfall = gained_experience - initial_experience + 1
                total_hours_needed += experience_shortfall
                # Update the hero's experience level after training.
                initial_experience += experience_shortfall

            # Deduct the energy used for this battle from the hero's current energy.
            initial_energy -= required_energy
            # Add the experience gained from this battle to the hero's experience.
            initial_experience += gained_experience

        # Return the total number of hours the hero needs to train to be able to defeat all enemies.
        return total_hours_needed

```

Java Solution

```java
class Solution {

    public int minNumberOfHours(int initialEnergy, int initialExperience, int[] energyNeeded, int[] experienceEarned) {
        int additionalHours = 0; // Stores the total additional hours of training required.

        // Loop through each training session
        for (int i = 0; i < energyNeeded.length; ++i) {
            int requiredEnergy = energyNeeded[i]; // Energy needed for the current training session.
            int requiredExperience = experienceEarned[i]; // Experience earned from the current training session.

            // If not enough energy, calculate and add the necessary training hours needed to gain energy.
            if (initialEnergy <= requiredEnergy) {
                additionalHours += requiredEnergy - initialEnergy + 1;
                initialEnergy = requiredEnergy + 1; // Update initialEnergy to the new value after training.
            }

            // If not enough experience, calculate and add the necessary training hours needed to gain experience.
            if (initialExperience <= requiredExperience) {
                additionalHours += requiredExperience - initialExperience + 1;
                initialExperience = requiredExperience + 1; // Update initialExperience to the new value after training.
            }

            // Deduct the used energy from the initialEnergy.
            initialEnergy -= requiredEnergy;
            // Add the gained experience to the initialExperience.
            initialExperience += requiredExperience;
        }

        // Return the total additional hours of training required.
        return additionalHours;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int minNumberOfHours(int initialEnergy, int initialExperience, vector<int>& energyRequired, vector<int>& experienceGained) {
        int additionalHours = 0; // Store the total additional hours needed

        // Iterating through each training session
        for (int i = 0; i < energyRequired.size(); ++i) {
            int energyNeeded = energyRequired[i]; // Energy needed for this session
            int experienceNeeded = experienceGained[i]; // Experience to be gained from this session

            // If initial energy is not enough, train to get just enough energy plus one
            if (initialEnergy <= energyNeeded) {
                additionalHours += energyNeeded - initialEnergy + 1; // Increment additional hours by the shortfall in energy plus or
                initialEnergy = energyNeeded + 1; // Update energy to the new level after training
            }

            // If initial experience is not enough, train to get just enough experience plus one
            if (initialExperience <= experienceNeeded) {
                additionalHours += experienceNeeded - initialExperience + 1; // Increment additional hours by the shortfall in experi
                initialExperience = experienceNeeded + 1; // Update experience to the new level after training
            }

            // After successful training or if already sufficient, reduce energy and increase experience for the current session
            initialEnergy -= energyNeeded; // Deduct the energy used for this session
            initialExperience += experienceNeeded; // Add the experience gained from this session
        }

        // Return the total additional hours of training needed to be ready for all sessions
        return additionalHours;
    }
};

```

Typescript Solution

```typescript
/**
 * Calculate the minimum number of training hours needed to beat all opponents.
 *
 * @param initialEnergy - The starting energy level of the player.
 * @param initialExperience - The starting experience points of the player.
 * @param energy - The array of energy points required to beat each opponent.
 * @param experience - The array of experience points the player gains after beating each opponent.
 * @returns The minimum number of training hours required.
 */
function minNumberOfHours(
    initialEnergy: number,
    initialExperience: number,
    energy: number[],
    experience: number[]
): number {
    const numberOfOpponents = energy.length;
    let totalTrainingHours = 0;

    for (let i = 0; i < numberOfOpponents; i++) {
        const opponentEnergy = energy[i];
        const opponentExperience = experience[i];

        // Check if the player's energy is less than or equal to the opponent's requirement
        if (initialEnergy <= opponentEnergy) {
            const energyShortage = opponentEnergy - initialEnergy + 1;
            totalTrainingHours += energyShortage;
            initialEnergy += energyShortage; // Increase player's energy to beat the opponent
        }

        // Check if the player's experience is less than or equal to the opponent's
        if (initialExperience <= opponentExperience) {
            const experienceShortage = opponentExperience - initialExperience + 1;
            totalTrainingHours += experienceShortage;
            initialExperience += experienceShortage; // Increase player's experience to beat the opponent
        }

        // After defeating the opponent, player spends energy and gains experience
        initialEnergy -= opponentEnergy;
        initialExperience += opponentExperience;
    }

    return totalTrainingHours;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is primarily determined by the loop that iterates over the two lists `energy` and `experience`. Since these lists are traversed in a single pass using `zip()`, the complexity depends on the length of the lists. Let's denote the length of the lists as `n`. Therefore, we can conclude that the time complexity is `O(n)`, where `n` is the number of battles (the length of the lists).

Space Complexity

The given code uses a few variables (`ans`, `initialEnergy`, `initialExperience`) but does not allocate any additional space that grows with the size of the input. The use of `zip()` does not create a new list but returns an iterator that produces tuples on demand, so it doesn't significantly affect the space complexity. As a result, the space complexity is `O(1)`, which means it is constant and does not depend on the size of the input lists.