1665. Minimum Initial Energy to Finish Tasks

Hard Greedy Array Sorting

Problem Description

In this task, we are given an array called `tasks`, with each element being a pair of values `[actual_i, minimum_i]`. Each pair represents a task where `actual_i` is the amount of energy needed to complete the `i`-th task and `minimum_i` is the minimum amount of energy required to start the task. The challenge is to figure out the least amount of energy we need initially to be able to complete all tasks, but what's unique is that we can tackle the tasks in any sequence we prefer.

Imagine having some tasks that need a significant amount of energy to start but consume less upon completion, while others are easy to start but require a lot of energy to complete. The optimal sequence to complete these tasks will ensure that you never find yourself short of energy before starting a task. Given this, we are asked to calculate the minimum initial energy.

Intuition

To arrive at an intuitive solution, consider that a task requiring a lot of energy to begin but not as much to finish should be tackled later than a task that consumes more energy than it requires to start. This is because finishing tasks that spend more energy than the minimum required will deplete the energy reserve, potentially preventing the start of tasks with high initial energy requirements.

Hence, the solution sorts the tasks by the difference between `actual_i` and `minimum_i`. The tasks with the smallest difference (even potentially negative) are placed first, since completing these tasks either depletes the least energy from the reserve or can even recharge it.

While iterating over the sorted tasks, the algorithm checks if the current energy level (`cur`) is sufficient to start the current task (`a`). If it's not enough, we need only increase our initial energy just enough to start the task (`a - cur`). We then deduct the actual energy spent to complete the task from `cur`. Throughout this process, `ans` tracks the total additional energy required, if any, which ultimately will be the minimum initial energy needed to complete all tasks.

Solution Approach

The solution makes use of the sort function, a basic algorithm in Python, to arrange the tasks such that the task with the smallest difference between the actual energy and the minimum required energy needed to start comes first. This is done to prioritize tasks that will either decrease our energy the least or could potentially increase it (if `actual_i < minimum_i`).

To understand this, let's assume we have a list of tasks, each represented as `[actual_i, minimum_i]`. We apply the `sorted` function with a custom sort key — the difference `x[0] - x[1]` (`actual_i - minimum_i`), which helps us to process tasks that are less energy-consuming first.

The data structure used here is a simple list of lists, a common choice for representing a sequence of elements where each element itself contains multiple pieces of data.

The algorithm follows these steps:

1. Initialize `ans` and `cur` to zero. `ans` will hold our final answer, and `cur` tracks the current energy level.
2. Sort the tasks by the difference (`a[0] - a[1]`), which is calculated for each task using a lambda function as the key in the `sorted` method.
3. Iterate through the sorted tasks list.
   - For each task, check if our current energy `cur` is less than the minimum required energy `m`.
   - If so, we update `ans` with the difference `m - cur`, as we need to boost our energy level up to the minimum required to start the task. We also update `cur` to reflect this new level of energy.
   - Deduct the actual amount of energy spent on the task `a` from `cur`.
4. The loop exits after all tasks have been processed, and `ans` now holds the minimum initial amount of energy we need to finish all the tasks without running out of energy at any point.

Using this approach allows us to guarantee that we always have enough energy to start each task by incrementing our initial energy only when necessary and by the smallest amount needed. After all tasks are completed, `ans` tells us the minimum energy we would need to reserve initially to accomplish this.

Example Walkthrough

Let's say we're given the following array of tasks: `[[3,1], [2,2], [1,2]]`. Each task is represented as `[actual_i, minimum_i]`.

1. We start by initializing `ans` and `cur` to zero. These variables will help us keep track of the additional energy needed (`ans`) and the current energy level (`cur`).
2. We need to sort the tasks by the difference between `actual_i` and `minimum_i`. Using a lambda function as the key in the `sorted` method, the tasks are sorted as follows:

   `[[1,2], [2,2], [3,1]]` → The task `[1,2]` has a difference of `-1` (it actually gives us energy), `[2,2]` breaks even, and `[3,1]` requires 2 units more energy than it gives back.
3. We iterate through the sorted tasks:
   - The first task is `[1, 2]`. `cur` is 0 and is less than the minimum required energy `2` for the task. We need to increase it by `2-0=2`, so `ans` becomes `2` and `cur` becomes `2`. After completing the task that consumes `1` unit of energy, `cur` is now `2-1=1`.
   - The next task is `[2, 2]`. `cur` is now `1` which is less than the `2` required to start the task. We must increase `cur` by `2-1=1`, which means `ans` is updated from `2` to `2+1=3`. After finishing the task that uses `2` units of energy, `cur` becomes `1-2=-1`. (This indicates that we finished the task using not just the current energy we had, but also some of the initial energy we reserved.)
   - The final task is `[3, 1]`. Our `cur` is `-1`, and since the minimum energy to start is `1`, we will boost `cur` by `1-(-1)=2`, making `ans` now `3+2=5`. We then complete the task by spending `3` units of energy, and `cur` is updated to `-1-3=-4`.
4. Having processed all tasks, `ans` holds the minimum initial amount of energy we need to complete all of them. In this case, we must start with at least `5` units of energy to ensure we can complete the tasks in this specific order.

By carefully selecting the sequence of the tasks and calculating the increments in our energy reserve only as needed, we find that starting with 5 units of energy allows us to complete all tasks successfully.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def minimumEffort(self, tasks: List[List[int]]) -> int:
        # Initialize the answer and the current energy to zero.
        energy_needed = current_energy = 0

        # Sort the tasks based on the difference between minimum energy needed
        # and actual energy consumed (i.e., sort by actual effort required).
        # We iterate through each sorted task.
        for actual, minimum in sorted(tasks, key=lambda x: x[0] - x[1]):
            # If current energy is less than the minimum energy needed for
            # the task, we need to increase our energy to at least the minimum.
            if current_energy < minimum:
                # Update the total energy needed by adding the difference
                # between the minimum energy needed and the current energy.
                energy_needed += minimum - current_energy
                # Set the current energy to the minimum required
                current_energy = minimum
            # After completing the task, decrease the current energy by
            # the amount of actual energy consumed.
            current_energy -= actual

        # Return the total energy needed to complete all tasks.
        return energy_needed

# Example usage:
# tasks = [[1, 2], [2, 4], [4, 8]]
# solution = Solution()
# print(solution.minimumEffort(tasks)) # Output will be the total minimum energy needed
```

Java

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

class Solution {

    // Calculates the minimum initial energy required to complete all tasks
    public int minimumEffort(int[][] tasks) {
        // Sort the tasks by the difference between the minimum energy to start and actual energy consumed
        Arrays.sort(tasks, (a, b) -> (b[1] - b[0]) - (a[1] - a[0]));

        int totalEnergyNeeded = 0; // Total initial energy needed to start all tasks
        int currentEnergy = 0;     // Current energy level while performing tasks

        // Iterate over each task to calculate the total initial energy required
        for (int[] task : tasks) {
            int actualEnergy = task[0];  // Actual energy spent to complete the task
            int minimumEnergyToStart = task[1]; // Minimum energy required to initiate the task

            // If current energy is less than the minimum required to start the task,
            // increase the total energy needed accordingly and update the current energy
            if (currentEnergy < minimumEnergyToStart) {
                totalEnergyNeeded += minimumEnergyToStart - currentEnergy;
                currentEnergy = minimumEnergyToStart;
            }

            // After task completion, reduce the current energy by the actual energy spent
            currentEnergy -= actualEnergy;
        }

        // Return the calculated total initial energy needed
        return totalEnergyNeeded;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // Needed to use the std::sort function

class Solution {
public:
    int minimumEffort(std::vector<std::vector<int>>& tasks) {
        // Sort the tasks based on the difference between the minimum initial energy required
        // and the actual energy consumed by the task. The task with the smallest difference
        // comes first because it requires less effort after completion to start the next task.
        std::sort(tasks.begin(), tasks.end(), [&](const auto& task1, const auto& task2) {
            return (task1[0] - task1[1]) < (task2[0] - task2[1]);
        });

        int totalEffort = 0; // Accumulated effort required to start all tasks
        int currentEnergy = 0; // Current available energy

        // Iterate through the sorted tasks
        for (auto& task : tasks) {
            int actualEnergyConsumption = task[0]; // Actual energy consumed by the task
            int minInitialEnergy = task[1]; // Minimum initial energy required to start the task

            // If currentEnergy is less than the minimum required initial energy
            // then increase the totalEffort by the difference and update currentEnergy
            if (currentEnergy < minInitialEnergy) {
                totalEffort += minInitialEnergy - currentEnergy;
                currentEnergy = minInitialEnergy;
            }

            // After the task is done, subtract the actual energy consumption from the
            // current energy to represent the new state of current energy
            currentEnergy -= actualEnergyConsumption;
        }

        // Return the accumulated totalEffort required to start all tasks
        return totalEffort;
    }
};
```

TypeScript

```typescript
function minimumEffort(tasks: number[][]): number {
    // Sort the tasks based on the difference between minimum initial energy and actual energy consumption
    // This helps to start with tasks that require less extra energy after considering their actual energy use
    tasks.sort((task1, task2) => (task1[0] - task1[1]) - (task2[0] - task2[1]));

    let totalMinimumEnergy = 0; // The accumulated energy needed to complete all tasks
    let currentEnergy = 0;      // Current energy level

    // Iterate over each task to calculate the minimum initial energy required to complete them
    for (const [actualEnergy, minimumInitialEnergy] of tasks) {
        // If the current energy is less than the task's minimum initial required energy
        if (currentEnergy < minimumInitialEnergy) {
            // Need to add the extra energy required to meet the minimum initial energy
            totalMinimumEnergy += minimumInitialEnergy - currentEnergy;
            // Update the current energy to match the minimum initial requirement
            currentEnergy = minimumInitialEnergy;
        }
        // After completing the task, the actual energy used is subtracted from the current energy
        currentEnergy -= actualEnergy;
    }

    // Return the accumulated minimum initial energy needed to complete all tasks
    return totalMinimumEnergy;
}
```

```python
from typing import List

class Solution:
    def minimumEffort(self, tasks: List[List[int]]) -> int:
        # Initialize the answer and the current energy to zero.
        energy_needed = current_energy = 0

        # Sort the tasks based on the difference between minimum energy needed
        # and actual energy consumed (i.e., sort by actual effort required).
        # We iterate through each sorted task.
        for actual, minimum in sorted(tasks, key=lambda x: x[0] - x[1]):
            # If current_energy is less than the minimum energy needed for
            # the task, we need to increase our energy to at least the minimum.
            if current_energy < minimum:
                # Update the total energy needed by adding the difference
                # between the minimum energy needed and the current energy.
                energy_needed += minimum - current_energy
                # Set the current energy to the minimum required
                current_energy = minimum
            # After completing the task, decrease the current energy by
            # the amount of actual energy consumed.
            current_energy -= actual

        # Return the total energy needed to complete all tasks.
        return energy_needed

# Example usage:
# tasks = [[1, 2], [2, 4], [4, 8]]
# solution = Solution()
# print(solution.minimumEffort(tasks)) # Output will be the total minimum energy needed
```

Time and Space Complexity

Time Complexity

The time complexity of the solution is dominated by the sorting of the tasks array and the subsequent iteration through the sorted tasks.

The sorting function uses the Timsort algorithm in Python, which has a time complexity of `O(n log n)` in the average and worst case, where `n` is the number of tasks. After sorting, the function goes through the tasks one by one, which takes `O(n)` time.

Therefore, combining both steps, the total time complexity of the function is `O(n log n)` due to the sorting step being the most significant factor.

Space Complexity

The space complexity of the solution can be considered as `O(1)` because the sorting is done in-place and no additional data structures that grow with the input size are used. The variables `ans`, `cur`, `a`, and `m` are all constant in space, requiring a fixed amount of memory regardless of the number of tasks.