1936. Add Minimum Number of Rungs

Medium Greedy Array

Problem Description

In this challenge, you are working with an array named `rungs`, which represents the heights of rungs on a ladder in strictly increasing order. Your starting position is on the floor, at height 0, and your goal is to reach the highest rung. However, there's a limitation: you can only move from your current position to the next rung if the height difference between them does not exceed a given integer `dist`. If the difference is more than `dist`, you're allowed to add new rungs at any positions as long as they are positive integer heights and not already present. Your task is to determine the minimum number of additional rungs you must add to the ladder to ensure you can reach the last rung.

Intuition

The solution employs a greedy strategy that focuses on minimizing the number of rungs that need to be added to reach the next rungable height. Since you can't jump up more than `dist` height from your current position, whenever the next rung is too high, you add as few rungs as possible to close the gap.

1. You start on the floor (height 0), which acts as the initial rung for calculation purposes, hence the code prepends `0` to the `rungs` list.
2. You then iterate through each pair of subsequent rungs (including the floor at height 0), calculating the difference in height between them.
3. If the difference between current and next rung is less than or equal to `dist`, you can climb without adding any new rungs.
4. If the difference is greater than `dist`, you find out how many rungs you need to insert between the two in order to make each step climbable. This can be calculated as `(b - a - 1) // dist`, where `a` is the height you're currently on, and `b` is the next rung's height. The `-1` is there since you can climb exactly `dist` height without inserting another rung.
5. The `//` operator performs integer division, which gives you the floor value needed since you can't add a fraction of a rung.

The sum of all the rungs that need to be added across the entire ladder gives us the final answer.

Solution Approach

The solution's implementation begins by artificially extending the `rungs` array with a `0` at the start. This is done to handle the initial step from the floor (height 0) to the first rung as part of the uniform process. The extension is reflected in this code snippet: `rungs = [0] + rungs`.

The next step in the approach is to iterate over each adjacent pair of rungs, including the starting point on the floor. The Python `pairwise` utility, which is not explicitly defined in the provided code snippet, likely generates tuples of two adjacent elements from the `rungs` list. If `pairwise` is not a built-in function or part of the standard Python library that you are using, you would need to implement a way to iterate over the rungs in pairs manually.

For every pair of rungs (denoted by `a` and `b`), we calculate the height difference, `b - a`, and then determine if a climb is possible without adding a new rung. If the height difference is greater than `dist`, then we need to add rungs.

To find the minimum number of rungs needed, the difference is divided by `dist`, and we perform integer division using `//`. Integer division is chosen because you can only add whole rungs, not fractions of a rung. Since we're interested in how many full `dist` gaps there are between `a` and `b`, subtracting 1 from the difference (`b - a - 1`) ensures we don't count an additional rung if `b - a` is an exact multiple of `dist`.

The summing of all these additional rungs needed to make each step climbable gives us the total number of rungs we need to add: `sum((b - a - 1) // dist for a, b in pairwise(rungs))`.

This implementation is simple and efficient, making it a classic example of a greedy algorithm, as it makes a locally optimal choice of adding rungs at each step without needing to reconsider these decisions later on. It does not use any complex data structures, relying instead on basic list operations and arithmetic to find the solution.

Example Walkthrough

Let's illustrate the solution approach with a simple example:

Assume `rungs = [1, 4, 7]` and `dist = 3`. We want to find the minimum number of additional rungs we need to add to this ladder so that the height difference between consecutive rungs is never more than 3.

1. We extend the `rungs` list by adding `0` at the beginning, resulting in `rungs = [0, 1, 4, 7]`.
2. We now iterate in pairs: (0, 1), (1, 4), (4, 7).
   - For the first pair (0, 1), the difference is `1 - 0 = 1`, which is less than or equal to `dist`, so no additional rungs are needed.
   - Moving to the second pair (1, 4), the difference is `4 - 1 = 3`. This is exactly `dist`, so again, no additional rungs are needed.
   - Examining the third pair (4, 7), the difference is `7 - 4 = 3`, which is equal to `dist`, so no new rungs are required.
3. In this case, at each step, the height difference does not exceed `dist`, so we don't need to add any additional rungs. The total number of additional rungs needed is `0`.

As we can see from this example, by working systematically through the ladder and applying the logic outlined in the solution approach, we can determine the minimum number of rungs to add to make the ladder climbable within the specified height difference. In this particular instance, no extra rungs are needed because the existing rungs already satisfy the condition that each step is climbable within the given `dist`.

Solution Implementation

Python

````python
from itertools import pairwise  # This is assuming you are using Python 3.10 or newer

class Solution:
    def addRungs(self, rungs: List[int], dist: int) -> int:
        # Add ground level (0) as the first rung for comparison
        rungs = [0] + rungs

        # Calculate the additional rungs required
        # Iterate over each pair of adjacent rungs
        additional_rungs = 0
        for lower_rung, higher_rung in pairwise(rungs):
            # Calculate the gap between the two rungs
            gap = higher_rung - lower_rung - 1
            # Divide gap by dist to find the number of additional rungs needed
            # We subtract 1 from the gap before division because if the distance is
            # just dist, we don't need an additional rung.
            # The ceiling of the division is obtained by using integer division
            # (//) after adding (dist - 1). This ensures that we always round up.
            additional_rungs += gap // dist

        return additional_rungs
```

If you are using a version of Python earlier than 3.10 and do not have the `pairwise` utility from `itertools`, you might need to

```python
def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = itertools.tee(iterable)
    next(b, None)
    return zip(a, b)

# And then the rest of the Solution class remains the same.
````

Java

```java
class Solution {
    public int addRungs(int[] rungs, int dist) {
        int additionalRungs = 0; // Initialize a variable to count additional rungs needed.
        int previousRungHeight = 0; // Initialize a variable to keep track of the height of the last rung.

        // Loop through the array of rungs.
        for (int currentRungHeight : rungs) {
            // Determine the gap between the current rung and the previous rung.
            int gap = currentRungHeight - previousRungHeight;

            // If the gap is larger than the distance `dist`, calculate how many additional rungs are needed.
            // We subtract 1 because if the gap is exactly equal to `dist` plus 1, no additional rung is needed.
            if (gap > dist) {
                additionalRungs += (gap - 1) / dist;
            }

            // Update the 'previousRungHeight' to the height of the current rung for the next iteration.
            previousRungHeight = currentRungHeight;
        }

        // Return the total number of additional rungs required to climb the ladder.
        return additionalRungs;
    }
}
```

C++

```cpp
#include <vector> // Include necessary header for using vectors

class Solution {
public:
    // This function calculates the minimum number of additional rungs required
    // to be able to climb a ladder where the maximum distance you can climb is 'dist'.
    // 'rungs' represents the heights at which the existing rungs are located.
    int addRungs(vector<int>& rungs, int dist) {
        int additionalRungs = 0; // Initialize the counter for additional rungs needed
        int previousRungHeight = 0; // Keep track of the previous rung's height, start from the ground level which is 0

        // Iterate through the vector of rungs
        for (int& currentRungHeight : rungs) {
            // Calculate the difference between the current rung height and the previous rung height
            // Then, find out how many rungs would fit in that distance, if needed
            additionalRungs += (currentRungHeight - previousRungHeight - 1) / dist;

            // Update the height of the previous rung to the current one for the next iteration
            previousRungHeight = currentRungHeight;
        }

        return additionalRungs; // Return the total count of additional rungs needed
    }
};
```

TypeScript

````typescript
function addRungs(rungs: number[], dist: number): number {
    let additionalRungs = 0; // Initialize count of additional rungs needed
    let previousRungHeight = 0; // Position of the last rung reached, start from the ground level (0)

    // Iterate over existing rungs to determine if additional rungs are needed
    for (const currentRungHeight of rungs) {
        // Calculate the number of additional rungs required for the current gap
        let gap = currentRungHeight - previousRungHeight - 1;
        additionalRungs += Math.floor(gap / dist); // Increase the additional rungs needed

        previousRungHeight = currentRungHeight; // Update the last rung reached
    }

    return additionalRungs; // Return the total number of additional rungs needed
}


from itertools import pairwise  # This is assuming you are using Python 3.10 or newer

class Solution:
    def addRungs(self, rungs: List[int], dist: int) -> int:
        # Add ground level (0) as the first rung for comparison
        rungs = [0] + rungs

        # Calculate the additional rungs required
        # Iterate over each pair of adjacent rungs
        additional_rungs = 0
        for lower_rung, higher_rung in pairwise(rungs):
            # Calculate the gap between the two rungs
            gap = higher_rung - lower_rung - 1
            # Divide gap by dist to find the number of additional rungs needed
            # We subtract 1 from the gap before division because if the distance is
            # just dist, we don't need an additional rung.
            # The ceiling of the division is obtained by using integer division
            # (//) after adding (dist - 1). This ensures that we always round up.
            additional_rungs += gap // dist

        return additional_rungs
```

If you are using a version of Python earlier than 3.10 and do not have the `pairwise` utility from `itertools`, you might need to eit

```python
def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = itertools.tee(iterable)
    next(b, None)
    return zip(a, b)

# And then the rest of the Solution class remains the same.
````

Time and Space Complexity

Time Complexity:

The time complexity of the given code is `O(n)`, where `n` is the number of elements in the 'rungs' list provided as input. The reasoning behind this is as follows:

- The expression `[0] + rungs` takes `O(n)` time since it creates a new list of size `n+1`.
- The `for a, b in pairwise(rungs)` loop iterates over the list of rungs, which, after adding the 0 at the beginning, has `n+1` elements. Since `pairwise` yields `n` pairs (since it considers adjacent pairs), the iteration takes place `n` times.
- Within each iteration, a subtraction `b - a`, a subtraction by 1 `(b - a - 1)`, and a floor division `// dist` occur. These operations are considered `O(1)`, as they take constant time regardless of the size of the numbers.
- The `sum` function iterates over the list of differences and adds them up, which is `O(n)` since it performs `n-1` addition operations.

Since all other operations are constant time, and the iteration is proportional to the size of the input list, the overall time complexity is `O(n)`.

Space Complexity:

The space complexity of the code is `O(1)`. Here's why:

- The additional list `[0] + rungs` does indeed create a new list, but this is part of the input transformation and is not considered in the space complexity analysis as it does not scale with the input size.
- The `pairwise(rungs)` function typically implements an iterator pattern which generates one pair at a time, rather than storing all pairs in memory. This implies constant additional space.
- Similarly, the floor division and subtraction operations within the summation do not require additional space that scales with the input size.
- The `sum` function aggregates the values into a single integer, which also takes constant space.

As such, no additional space is required that grows with the size of the input, which leads to a space complexity of `O(1)`.