1883. Minimum Skips to Arrive at Meeting On Time

Hard | Array | Dynamic Programming | Leetcode Link

Problem Description

You are scheduled to attend a meeting and have a fixed number of hours, `hoursBefore`, to cover a certain distance through `n` roads. The lengths of these roads are represented as an integer array `dist`, where `dist[i]` is the length of the `i`th road in kilometers. You are given a consistent speed in km/h, denoted by the variable `speed`.

There is a constraint imposed on your travel -- after completing each road, you're required to rest until the start of the next whole hour before you commence the next leg of your journey. However, this mandatory rest period is waived for the final road as you've already arrived at your destination by then.

Interestingly, to help you arrive on time, you are permitted to skip some of these rest periods. Skipping a rest means you don't have to wait for the next whole hour, which might help you align your travel more closely with the hours available.

The core challenge of the problem is determining the minimum number of rest periods you must skip to ensure you arrive at your meeting on time, if at all possible. If there's no way to make it on time, you should return `-1`.

Intuition

To solve this problem, dynamic programming is an appropriate approach because we can break down the larger problem into smaller subproblems. Each subproblem involves evaluating the minimal time required to reach the end of a certain road with a specific number of skips.

The intuition behind this approach begins with examining every possible scenario where you choose to either wait for the rest period or skip it after each road. We aim to find the strategy that takes the least amount of time to arrive at the meeting on time while skipping the fewest rest periods.

We maintain an array `f`, where `f[i][j]` represents the minimum time to reach the end of the `i`th road with `j` skips. The algorithm iterates through each road and each possible number of skips, updating the time calculation to reflect whether we wait or skip after each road.

By initializing `f[i][j]` with an infinitely large number, we ensure that any actual calculated time will be considered in our comparisons. The iterative process compares the current minimum time `f[i][j]` with the time achieved by either waiting or skipping the rest period, updating as needed to hold the least time.

With all the subproblems solved, we look for the minimum number of skips needed that still allows us to arrive within `hoursBefore`. We iterate through the possible number of skips and return the smallest number for which the corresponding time does not exceed `hoursBefore`. If all scenarios exceed the time limit, we return `-1`, signaling it's impossible to attend the meeting on time.

The provided solution code represents this approach and calculates the minimum skips required accurately using dynamic programming.

Solution Approach

The solution utilizes dynamic programming, which involves breaking down the problem into smaller, manageable subproblems and then building up to the solution.

Data Structures:

- A 2D list `f` of size `(n + 1) x (n + 1)` is initialized with `inf` (representing infinity). This array will store the minimum time needed to reach each road with a given number of skips.

Algorithms and Patterns:

1. **Initialization**: Since Python lists start at index 0 but roads start at 1, we set `f[0][0]` to 0 to denote that at the beginning (before traveling any road), no time has passed.
2. **Dynamic Programming Iteration**:
   - The outer loop iterates through each road (denoted by `i`), starting from 1, since we consider 0 as the starting point before traveling any road.
   - The inner loop iterates through every possible number of skips (`j`) that could be applied up to the current road `i`.
3. **State Transition**:
   - Inside the inner loop, the algorithm considers two scenarios to find the minimum time to complete road `i` with `j` skips:
     - **Waiting**: The algorithm calculates the time if we wait for the next integer hour mark after road `i-1`. This is done by adding `x / speed` (time to travel the current road) to `f[i - 1][j]` (minimum time to reach the previous road with the same number of skips) and applying `ceil` to round up to the next hour. An epsilon `eps` is subtracted before applying the `ceil` function to handle floating point precision issues.
     - **Skipping**: The time is directly added without the ceiling for the case where we skip the rest period after road `i-1`. This is only applicable if `j` is non-zero, i.e., we have skipped at least once.
4. **Comparing and Updating States**:
   - The algorithm compares the outcomes of waiting and skipping, updating `f[i][j]` with the minimum value.
5. **Finding the Result**:
   - After we've filled in the `f` array with the minimum traveling times for all possible skips for each road, the algorithm looks for the smallest number of skips that would still allow us to arrive within `hoursBefore`.
   - This is done by iterating over the possible number of skips for the final road (`f[n][j]` where `j` ranges from `0` to `n`) and checking if that time is less than or equal to `hoursBefore`. A small epsilon `eps` is added to the time limit to handle floating point precision issues.
   - If such a `j` is found, it's returned as the minimum number of skips required. If all possible skips result in times greater than `hoursBefore`, the function returns `-1`.

By the end of the iteration, the solution will have considered every possible way to allocate the skips to minimize the travel time and determined whether it is possible to arrive on time.

Mathematical Formulas:

- The `ceil` function is used to account for the need to round up to the next integer hour when waiting after each road, and subtraction of an epsilon is used to correct for any floating point errors that could incorrectly round down when right on the hour mark.

The algorithm's complexity is bounded by the number of roads `n` and the number of possible skips, which can also be at most `n`. Hence, we have a time complexity of $O(n^2)$, which is generally acceptable for the road lengths and number of hours before the meeting typically encountered in this type of problem.

Example Walkthrough

To illustrate the solution approach, let's take a small example. Assume we need to cover a distance through `3` roads with lengths given by the array `dist = [5, 3, 6]`, where `hoursBefore = 6` hours and our consistent speed is `speed = 1` km/h.

Let's create an array `f` of size `(n + 1) x (n + 1)` filled with `inf` to represent the minimum time taken with a corresponding number of skips. The `f` array would be a 4×4 array in this case, considering `n = 3`.

We first initialize `f[0][0] = 0`. This denotes that we're at the starting point and haven't spent any time or skips yet.

The possible states as we iterate through our array `f`:

```
f[0][0] = 0
```

Moving on to the first road of 5 km:

- Without skipping, we'll spend `ceil(5 / 1) = 5` hours, so `f[1][0] = 5`.
- Skipping is not an option since it's the first road.

```
f[0][0] = 0
f[1][0] = 5
```

Next, the second road of 3 km:

- Without skipping, we'll spend `ceil((5 + 3) / 1) = 8` hours in total (`5` from the first road, and `3` for this road), so `f[2][0] = 8`.
- With one skip (assuming we skipped after finishing the first road), we would spend `5 (from the first road without waiting) + 3 (second road)` hours, so `f[2][1] = 8`.

```
f[0][0] = 0
f[1][0] = 5
f[2][0] = 8
f[2][1] = 8
```

Finally, the third road of 6 km:

- Without skipping, we're now at `ceil((8 + 6) / 1) = 14` hours, but since we can't reach our destination on time without skipping, this attempt is infeasible, so we skip considering `f[3][0]`.
- With one skip carried over and skipping after the second road, we would spend `5 + 3 + 6 = 14` hours, which is again infeasible, so `f[3][1]` doesn't need to be considered.
- With two skips, assuming we skipped after the first and second roads, we would spend `5 + 3 + 6 = 14` hours. Since we can skip the rest on the last road, `f[3][2] = 14`.

```
f[0][0] = 0
f[1][0] = 5
f[2][0] = 8 (infeasible as exceeds `hoursBefore`)
f[2][1] = 8 (infeasible as exceeds `hoursBefore`)
f[3][0] = x
f[3][1] = x
f[3][2] = 14
```

Now, we must consider all options for the number of skips (up to the number of roads `n=3`). Ideally, we would look for `f[n][j]` where `j` ranges from `0` to `n` (3 in this case) such that `f[n][j] <= hoursBefore (6)`.

When evaluating `f[3][2]`, we can see the value is `14` which exceeds `hoursBefore`. Thus, none of our options meet the requirement, and we conclude that it is impossible to arrive on time. We return `-1`.

Python Solution

```python
from math import ceil

class Solution:
    def minSkips(self, distances: List[int], speed: int, hours_before: int) -> int:
        # Length of the distances array
        num_of_sections = len(distances)

        # Initialize a DP table with infinity to store number of skips for reaching certain points
        dp = [[float('inf')] * (num_of_sections + 1) for _ in range(num_of_sections + 1)]

        # At the beginning, zero skips are made and the total time is also zero
        dp[0][0] = 0

        # A very small epsilon value to avoid floating-point precision issues
        epsilon = 1e-8

        # Iterate through each section of the distances
        for i, distance in enumerate(distances, 1):
            # For each section, calculate the minimum number of skips required to reach it
            for skips in range(i + 1):
                if skips < i:
                    # Option 1: Do not skip this section but might need to skip a future section
                    # Ceil is used to round up to the nearest integer to enforce a rest period
                    dp[i][skips] = min(dp[i][skips], ceil(dp[i - 1][skips] + distance / speed - epsilon))
                if skips:
                    # Option 2: Skip this section and therefore no rest needed (no ceil)
                    # Can only skip if skips > 0, hence we check it
                    dp[i][skips] = min(dp[i][skips], dp[i - 1][skips - 1] + distance / speed)

        # Check each possible number of skips to see if it fits within the hours_before limit
        for skips in range(num_of_sections + 1):
            if dp[num_of_sections][skips] <= hours_before + epsilon:
                return skips

        # If none fit within the hours_before limit, return -1
        return -1

```

Java Solution

```java
class Solution {
    public int minSkips(int[] distances, int speed, int hoursBefore) {
        int numSegments = distances.length;
        double[][] dp = new double[numSegments + 1][numSegments + 1];

        // Initialize the dp array with a high value
        for (int i = 0; i <= numSegments; i++) {
            Arrays.fill(dp[i], 1e20);
        }

        // Base case: no distance covered without any skips
        dp[0][0] = 0;

        // Small value to avoid precision issues
        double eps = 1e-8;

        // Populate the dp array
        for (int i = 1; i <= numSegments; ++i) {
            for (int j = 0; j <= i; ++j) {
                // If not taking the maximum number of skips (which would be the segment index)
                if (j < i) {
                    dp[i][j] = Math.min(
                        dp[i][j],
                        Math.ceil(dp[i - 1][j] - eps) + (double) distances[i - 1] / speed
                    );
                }
                // If taking one less skip
                if (j > 0) {
                    dp[i][j] = Math.min(
                        dp[i][j],
                        dp[i - 1][j - 1] + (double) distances[i - 1] / speed
                    );
                }
            }
        }

        // Determine the minimum number of skips needed to reach the target within hoursBefore
        for (int j = 0; j <= numSegments; ++j) {
            if (dp[numSegments][j] <= hoursBefore + eps) {
                return j; // Found the minimum skips required
            }
        }

        return -1; // It's not possible to reach on time with given constraints
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int minSkips(vector<int>& distances, int speed, int hoursBefore) {
        // Number of different distances that we need to travel.
        int numDistances = distances.size();

        // 'dp' will store the minimum time required to reach a certain point with a given number of skips.
        // Initialize all values to a very high number, interpreted as 'infinity'.
        vector<vector<double>> dp(numDistances + 1, vector<double>(numDistances + 1, 1e20));

        // Base case: the time to reach the starting point with 0 skips is 0.
        dp[0][0] = 0;

        // A small epsilon is used to perform accurate floating-point comparisons later on.
        double eps = 1e-8;

        // Iterate over each segment of the journey.
        for (int i = 1; i <= numDistances; ++i) {
            // Iterate over the possible number of skips.
            for (int j = 0; j <= i; ++j) {
                // If we are not skipping the current segment.
                if (j < i) {
                    dp[i][j] = min(dp[i][j], ceil(dp[i - 1][j] + static_cast<double>(distances[i - 1]) / speed - eps));
                }

                // If we decide to skip a resting time period.
                if (j > 0) {
                    dp[i][j] = min(dp[i][j], dp[i - 1][j - 1] + static_cast<double>(distances[i - 1]) / speed);
                }
            }
        }

        // Look for the smallest number of skips needed to arrive before or exactly at the prescribed time.
        for (int j = 0; j <= numDistances; ++j) {
            if (dp[numDistances][j] <= hoursBefore + eps) {
                return j;  // The minimum number of skips required.
            }
        }

        // If it is not possible to arrive on time, return -1.
        return -1;
    }
};

```

Typescript Solution

```typescript
function minSkips(distances: number[], speed: number, hoursBefore: number): number {
    // Number of road segments
    const numSegments = distances.length;

    // Create a 2D array to record the minimum time needed to skip certain number of rests
    const minTime = Array.from({ length: numSegments + 1 },
                               () => Array.from({ length: numSegments + 1 },
                               () => Infinity));

    // Base case: starting with no distance and no skips, so time is 0
    minTime[0][0] = 0;

    // Small epsilon value to avoid precision issues with floating point numbers
    const eps = 1e-8;

    // Iterate through each segment
    for (let i = 1; i <= numSegments; ++i) {
        // Calculate the minimum time required with available number of skips
        for (let j = 0; j <= i; ++j) {
            // If we do not skip the current rest, use ceil to round to the next integer if necessary
            if (j < i) {
                minTime[i][j] = Math.min(minTime[i][j],
                                         Math.ceil(minTime[i - 1][j] + distances[i - 1] / speed - eps));
            }

            // If we can skip, use the exact value (no need to round)
            if (j) {
                minTime[i][j] = Math.min(minTime[i][j],
                                         minTime[i - 1][j - 1] + distances[i - 1] / speed);
            }
        }
    }

    // Check the minimum number of skips needed to reach within hoursBefore
    for (let j = 0; j <= numSegments; ++j) {
        if (minTime[numSegments][j] <= hoursBefore + eps) {
            return j; // Returns the minimum number of skips needed
        }
    }

    // If impossible to reach within hoursBefore, return -1
    return -1;
}

```

Time and Space Complexity

The time complexity of the provided code is `O(n^3)`, where `n` is the length of the `dist` array. This is because there are three nested loops: the outermost loop iterating through each element in `dist`, and two inner loops, where one loop iterates through `j` from `0` to `i+1`, and the other considers two possible scenarios for each `j` (either skipping or not skipping). Each operation inside the innermost loop is `O(1)`, but since the loops are nested, the total time complexity scales cubically.

The space complexity of the code is `O(n^2)` because of the 2D list `f` that has dimensions `n+1` by `n+1`. This list stores the minimum time required to reach each point with a given number of skips. Since it needs to hold a value for every combination of distances and skips, the space required grows quadratically with the number of distances.