1870. Minimum Speed to Arrive on Time

Medium Array Binary Search Leetcode Link

Problem Description

You have a specific amount of time, `hour`, to commute to the office using a series of trains. Each segment of your journey has a fixed distance, and all segments must be completed sequentially. The key challenge is that trains can only leave at the start of each hour, meaning you might need to wait before boarding the next train. For instance, if a train ride takes 1.5 hours, a 0.5-hour wait is required before the next train departs. Your task is to determine the minimum train speed (in km/h) required to reach the office on time. This speed must be a positive integer, and if it's impossible to arrive on time, the function should return `-1`. The provided constraints ensure that the solution won't be higher than `10^7` and the `hour` won't have more than two decimals.

Intuition

The minimum speed to reach on time is not evidently clear and can range from 1 to `10^7`. To find the minimum speed efficiently, we employ a binary search strategy. A binary search halves the potential search space by checking whether a particular condition (in this case, being able to make it to the office on time at a certain speed) is met or not.

Here's the thought process for arriving at the binary search solution:

1. If a particular speed allows us to reach the office on time (`hour` or less), then any speed higher than this would also suffice. Conversely, any speed lower than a speed that does not make it on time is also inadequate.

2. We define a `check` function to determine if the current speed is sufficient to reach on time, considering the waiting time between train rides. The last ride doesn't require any wait time.

3. Using a binary search, we find the minimum speed at which the `check` function returns `true`. As speeds lower than this minimum won't help us arrive on time, we restrict our search to the range where we might meet the condition.

4. We initiate a search range from 1 to `10^7` and repeatedly narrow it down using the `check` function until we pinpoint the minimum required speed.

5. If we're unable to find a speed that allows for a timely arrival within the maximum speed limit, we return `-1`, indicating that it's not possible to be punctual.

The binary search is implemented using the `bisect_left` function from the Python library. It operates by taking the range as an iterable, the condition to check, and a key function, which in this case is the `check` function. It then finds the leftmost value in the sorted range that satisfies the condition.

Solution Approach

The solution to the problem leverages the binary search algorithm, which is ideal for the scenario because of its ability to reduce the search space in half with each iteration. The objective of the binary search in this context is to discover the minimum integer speed so that the total travel time does not exceed the given `hour`.

Here's how the binary search is used to implement the solution:

- **Define a `check` function:** This function accepts a speed and calculates the total time taken to travel all distances in the `dist` array at that speed. For all train segments except the last, the time is rounded up to the nearest integer to account for the rule that trains leave only at the beginning of every hour. For the last train segment, exact time can be used since you don't need to wait for another train after.

- **Binary search algorithm:** We perform a binary search to find the appropriate speed using two boundaries denoted as `left` and `right`, initially set to `1` and `10^7 + 1` respectively. The algorithm iteratively narrows these boundaries until the optimal speed is found.

  - **Initialization:** We set our initial search interval from `1` to `10^7 + 1`, to cover the entire range of possible speeds as per the problem's constraints.

  - **Binary search loop:** At each step of the loop, the midpoint of the current interval is computed and passed to the `check` function. Based on the function's output:

    - If the check is `true`, it means the current speed or any higher could be the solution; hence we narrow the search to the lower half of the interval (`right = mid`).

    - If the check is `false`, it means the current speed is too slow, so we narrow the search to the upper half of the interval (`left = mid + 1`).

  - **Checking for impossibility:** If it's determined that even the highest possible speed doesn't allow us to be on time (at after searching the entire interval), we conclude that it's impossible to reach on time and return `-1`.

- **Using `bisect_left`:** The binary search is succinctly executed using the `bisect_left` function from the `bisect` module, which is applied to a range object representing our potential speeds. We use a lambda function as the key to the `bisect_left`, which internally applies the `check` function to the speeds to handle the binary search. The result is then adjusted by adding `1` to the speed since `bisect_left` finds the leftmost place where `True` could be inserted without changing the order.

- **Mathematical and logical operations:** The algorithm uses integer division (`math.ceil`) to account for the waiting time at stations and leverages boolean conditions to implement the binary search logic.

To sum it up, the solution efficiently navigates through a potentially massive search space to find the minimum speed necessary for timely arrival, using binary search principles encapsulated in Python's `bisect_left` function.

Example Walkthrough

Suppose we have the following input parameters:

- Distances array `dist = [5, 7, 3]`, representing the distances of each train segment of the journey.
- Total time allowed `hour = 2.5` hours to reach the office.

Now, let's walk through the binary search approach to find the minimum train speed required for the given journey:

1. **Initialization**: We start with a potential speed range of `1` to `10^7 + 1`. This means our `left` boundary is `1` and our `right` boundary is `10^7 + 1`.

2. **First iteration of binary search**:

   - The mid-speed is `(1 + 10000001) / 2 = 5000001`.
   - We check if a train at 5000001 km/h can reach within `2.5` hours.
   - We calculate the time for each segment at `5000001` km/h:
     - Segment 1: `5 / 5000001` hours - No need to round up because there is another segment after.
     - Segment 2: `7 / 5000001` hours - No need to round up because there is another segment after.
     - Segment 3: `3 / 5000001` hours - This is the last segment, we take the exact time.
   - We sum up these times and find that the total time taken is way less than `2.5` hours, so we set `right` to `5000001`.
3. **Second iteration of binary search**:

   - The new mid-speed is `(1 + 5000001) / 2 = 2500001`.
   - We perform the same `check` function for the new mid-speed.
   - Again, the speed is high enough to get us there in under `2.5` hours, so we adjust `right` to `2500001`.
4. **Continuing the binary search**:

   - We continue halving the interval and checking, bringing the `left` and `right` boundaries closer after each iteration.
5. **Final step**:

   - Eventually, we narrow down to a range where moving `left` by one would cause `check` to yield `false`, and we settle on the `right` boundary as the minimum sufficient speed.

   - Suppose after multiple iterations, we find that at `10` km/h, `check` is `false`, but at `11` km/h, `check` is `true`. This tells us that `11` km/h is the minimum speed required to reach the office on time.

6. **Result**:

   - The binary search terminates when `left` is equal to `right`, which means we've found the minimum train speed satisfying the conditions.
   - If no such speed allows arrival within `2.5` hours, our search boundaries would converge such that `check` always returns `false`, and we return `-1`.

In our example, by following the binary search algorithm steps, we would eventually find that the minimum speed required for the distances `[5, 7, 3]` to be covered within `2.5` hours is `11` km/h. This result is achieved without testing every speed from `1` to `10^7` individually.

Python Solution

```python
from bisect import bisect_left
import math
from typing import List

class Solution:
    def minSpeedOnTime(self, dist: List[int], hour: float) -> int:
        # Helper function to check if a given speed is sufficient to arrive on time
        def is_speed_sufficient(speed):
            total_time = 0
            for i, distance in enumerate(dist):
                # For the last distance, we don't need to ceil as we can arrive exactly on time
                if i == len(dist) - 1:
                    total_time += distance / speed
                else:  # For all others, we'll ceil to account for the fact that we can't travel partial units of distance in less th
                    total_time += math.ceil(distance / speed)
            return total_time <= hour

        # Set the maximum possible speed based on constraints; here, arbitrarily set to 10^7
        max_possible_speed = 10**7 + 1

        # Binary search to find the minimum sufficient speed [1, max_possible_speed)
        # We are adding 1 because `bisect_left` will return the position to insert True to maintain sorted order
        # So we need to convert this position to the corresponding speed by adding 1
        minimum_sufficient_speed = bisect_left(range(1, max_possible_speed), True, key=is_speed_sufficient) + 1

        # If the speed is equal to the maximum possible speed, it means it wasn't possible to arrive on time
        # So we return -1; otherwise, return the minimum sufficient speed
        return -1 if minimum_sufficient_speed == max_possible_speed else minimum_sufficient_speed

```

Java Solution

```java
class Solution {

    public int minSpeedOnTime(int[] distances, double hour) {
        int lowerBound = 1; // Defines the minimum possible speed
        int upperBound = (int) 1e7; // Defines the maximum possible speed, assuming a constraints' defined upper limit

        // Binary search to find minimum speed necessary to arrive on time
        while (lowerBound < upperBound) {
            int midSpeed = (lowerBound + upperBound) / 2; // Use mid as the candidate speed
            if (canArriveOnTime(distances, midSpeed, hour)) {
                upperBound = midSpeed; // If we can arrive on time with this speed, try lower speed
            } else {
                lowerBound = midSpeed + 1; // Otherwise, try a higher speed
            }
        }

        // Check if the minimum speed found allows arrival on time
        return canArriveOnTime(distances, lowerBound, hour) ? lowerBound : -1;
    }

    // Helper function to check if we can arrive on time given the distances, speed and hour
    private boolean canArriveOnTime(int[] distances, int speed, double hour) {
        // Total time taken to travel all distances at the given speed
        double totalTime = 0.0;
        for (int i = 0; i < distances.length; ++i) {
            double segmentTime = (double)distances[i] / speed;
            // Ceil the time for all segments except the last (no need to wait for a whole hour on the last segment)
            totalTime += (i == distances.length - 1) ? segmentTime : Math.ceil(segmentTime);
        }
        return totalTime <= hour; // Return true if time does not exceed the given hour
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the minimum speed needed to arrive on time
    int minSpeedOnTime(vector<int>& distances, double hour) {
        // Initialize binary search bounds
        int minSpeed = 1, maxSpeed = 1e7;

        // Perform binary search to find the minimum feasible speed
        while (minSpeed < maxSpeed) {
            int midSpeed = (minSpeed + maxSpeed) >> 1; // Calculate mid speed

            // Check if current speed meets the required time
            if (canArriveOnTime(distances, midSpeed, hour)) {
                maxSpeed = midSpeed; // If yes, search in the lower half
            } else {
                minSpeed = midSpeed + 1; // If no, search in the upper half
            }
        }

        // After binary search, check if the left bound allows to arrive on time
        return canArriveOnTime(distances, minSpeed, hour) ? minSpeed : -1;
    }

    // Helper function to check if it is possible to arrive on time at the given speed
    bool canArriveOnTime(vector<int>& distances, int speed, double hour) {
        double totalTime = 0; // Variable to store total time taken

        for (int i = 0; i < distances.size(); ++i) {
            double travelTime = static_cast<double>(distances[i]) / speed;

            // For all but the last distance, we round up the travel time to the nearest whole number
            // since you can't travel a fraction of a distance without spending the whole hour
            totalTime += (i == distances.size() - 1) ? travelTime : ceil(travelTime);
        }

        // Return true if the total time does not exceed the given hour, false otherwise
        return totalTime <= hour;
    }
};

```

Typescript Solution

```typescript
/**
 * Calculate the minimum travel speed required to complete a given set of distances within a specified time.
 *
 * @param {number[]} distances - An array of distances to travel.
 * @param {number} timeLimit - The time limit to complete all travels.
 * @return {number} - The minimum speed required to be on time, or -1 if it's impossible.
 */
function minSpeedOnTime(distances: number[], timeLimit: number): number {
    // Check if the travel is possible within the time limit. If there are more distances
    // than the ceiling value of hours, it's impossible to complete on time.
    if (distances.length > Math.ceil(timeLimit)) return -1;

    let minSpeed = 1; // Lower bound for binary search (minimum possible speed).
    let maxSpeed = 10 ** 7; // Upper bound for binary search (arbitrarily high speed).

    while (minSpeed < maxSpeed) {
        let midSpeed = Math.floor((minSpeed + maxSpeed) / 2);
        if (arriveOnTime(distances, midSpeed, timeLimit)) {
            maxSpeed = midSpeed;
        } else {
            minSpeed = midSpeed + 1;
        }
    }

    // The left boundary of the binary search represents the minimum speed at which we can arrive on time.
    return minSpeed;
}

/**
 * Helper function to check if it's possible to arrive on time at the given speed.
 *
 * @param {number[]} distances - An array of distances to travel.
 * @param {number} speed - The traveling speed.
 * @param {number} timeLimit - The time limit to complete all travels.
 * @return {boolean} - Returns true if it's possible to arrive on time, otherwise false.
 */
function arriveOnTime(distances: number[], speed: number, timeLimit: number): boolean {
    let totalTime = 0.0;
    let n = distances.length;

    for (let i = 0; i < n; i++) {
        let travelTime = distances[i] / speed;
        // For all but the last distance, round the travel time up to the nearest whole number,
        // since you can't travel a fraction of the distance at a consistent speed.
        if (i != n - 1) {
            travelTime = Math.ceil(travelTime);
        }
        totalTime += travelTime;
    }
    // Compare the total time taken to travel at the given speed with the time limit.
    return totalTime <= timeLimit;
}

```

Time and Space Complexity

The time complexity of the `minSpeedOnTime` function is determined by the binary search and the check function that is called at each step of the binary search.

The binary search runs in `O(log R)`, where `R` is the range of possible speeds, which in this case is `10^7`. The `+1` correction does not affect the logarithmic complexity.

The `check` function runs in `O(N)`, where `N` is the number of elements in the `dist` list because it needs to iterate over all the distances. Inside the check function, `math.ceil` function is called which has a constant time `O(1)`. Therefore, for each call of the `check` function, the time complexity is linear with respect to the number of distances.

Therefore, the overall time complexity of the function is `O(N log R)` where `N` is the number of distances in the `dist` list and `R` is `10^7`.

The space complexity of the function is `O(1)`. No additional space is allocated that grows with the size of the input, except for the variable `res`, which uses a constant amount of space.