2187. Minimum Time to Complete Trips

Medium Array Binary Search

Problem Description

In this problem, we are given an array `time` that represents the time each bus takes to complete a single trip. Importantly, the buses can each make as many trips as needed back-to-back, without any downtime in between successive trips. Moreover, the buses operate independently from one another, meaning that the trips one bus makes have no effect on another's trips.

We also receive an integer `totalTrips`, which signifies the collective number of trips that all buses combined must complete. The task is to determine the minimum amount of time required for all buses to collectively complete at least `totalTrips` trips. Keep in mind that it's the combined trips of all buses that we are interested in, not just the trips made by one bus.

Intuition

The solution revolves around the concept of binary search. Initially, we can think that the minimum time required for all buses to complete the required number of trips would be if all buses operated at their fastest possible times, which is why we use the minimum bus time multiplied by the `totalTrips` as an upper bound for the time needed.

Given that the buses operate independently, we need to find the point at which the sum of trips made by all buses hits at least `totalTrips`. To do this efficiently, we use binary search instead of checking each possible time sequentially.

Binary search works here because as time increases, the number of trips that can be completed is non-decreasing - this is a monotonic relationship, which is a requirement for binary search to be applicable.

The key insight behind the solution is that for a given amount of time `t`, we can calculate the total number of trips completed by all buses by summing up `t // time[i]` for each bus `i`. If this total number of trips is less than `totalTrips`, we know we need more time, and if it's more, we have a potential solution but we'll continue to search for a smaller time value that still meets the `totalTrips` requirement.

The `bisect_left` function from Python's standard library is then used to find the smallest `t` where the calculated number of trips is at least `totalTrips`. It does this by looking for the leftmost insertion point to maintain sorted order.

Solution Approach

The solution leverages binary search, an efficient algorithm for finding an item from a sorted collection by repeatedly halving the search interval. In this problem, although the starting `time` array is not inherently sorted, we are searching through a range of time from `0` to `mx`, where `mx` is the product of the minimum value in the `time` array and `totalTrips`. This range of time is conceptually sorted because, as mentioned earlier, the number of total trips completed by all buses increases monotonically as time increases.

The Python `bisect_left` function is used in this context to perform the binary search. The `bisect_left` function requires a range or a sorted list to search within, a target value to find, and an optional `key` function to transform the elements before comparison.

In this problem, our `range` represents all possible times from `0` to `mx`. The target we are seeking is `totalTrips`, our criterion for the minimum trips required.

The `key` function is particularly significant here. For every mid-value time `x` during the binary search, the `key` function calculates the total number of trips that all buses would have completed by that time using the expression `sum(x // v for v in time)` where `v` is each individual time from the `time` list. The `//` operator performs integer division, yielding the number of trips a single bus completes in time `x`.

The process goes as follows:

- If the calculated total trips for the `mid` time value is less than `totalTrips`, it means more time is needed for the buses to complete the required number of trips, so the search continues to the right half of the current range.
- If the total is greater or equal, the `mid` value is a potential solution, but we check to the left half of the range to find if there is a smaller viable time.

This continues until the binary search narrows down to the minimum time where the buses can collectively complete `totalTrips`. That is the value returned by the `bisect_left` function as a result, hence providing us with the minimum time needed to achieve at least `totalTrips` trips by all buses.

Here is the important part of the solution code for reference, focusing on the applied binary search mechanism:

```
class Solution:
    def minimumTime(self, time: List[int], totalTrips: int) -> int:
        mx = min(time) * totalTrips
        return bisect_left(
            range(mx), totalTrips, key=lambda x: sum(x // v for v in time)
        )
```

No additional data structures are needed beyond the input `time` list and the range object created for the search interval. The binary search pattern itself acts as the algorithmic data structure guiding the search for the correct minimum time.

Example Walkthrough

Let's walk through a simple example to illustrate the solution approach.

Suppose `time = [3, 6, 8]` represents the time taken by three buses to complete a single trip. We need to find the minimum amount of time required for these buses to complete at least `totalTrips = 7` trips in total.

First, we determine an upper bound for the binary search based on the fastest bus. The minimum time in `time` is `3`, so the maximum conceivable time would be `3 * 7 = 21`. This is because if only the fastest bus were running, it would take 21 units of time to complete 7 trips.

Next, we use binary search to find the minimum time. The range for binary search is from `0` to `21`:

1. In the first iteration, the midpoint (`mid`) of `0` and `21` is `10`. We check how many total trips can be completed by all the buses in `10` units of time.
   - Bus 1 can make `10 // 3 = 3` trips.
   - Bus 2 can make `10 // 6 = 1` trip.
   - Bus 3 can make `10 // 8 = 1` trip.
   - So, the total is `3 + 1 + 1 = 5` trips, which is less than `7`. We need more time.
2. The next `mid` will be the midpoint between `11` and `21`, which is `16`.
   - Bus 1 can make `16 // 3 = 5` trips.
   - Bus 2 can make `16 // 6 = 2` trips.
   - Bus 3 can make `16 // 8 = 2` trips.
   - Now, the total is `5 > 3 + 2 + 1` trips, which is more than `7`. We have a potential solution but we'll continue to search in case there's a lower value that also works.
3. The next midpoint will be between `11` and `15`, giving us `13`.
   - Bus 1 can make `13 // 3 = 4` trips.
   - Bus 2 can make `13 // 6 = 2` trips.
   - Bus 3 can make `13 // 8 = 1` trip.
   - The total is `4 + 2 + 1 = 7` trips, which is exactly what we need.
4. Even though we've found that `13` minutes would work, we need to make sure there isn't a smaller value that also meets the `totalTrips` requirement, so we look to the left, between `11` and `12`.
5. As we proceed with the search, we find that `12` minutes won't suffice for 7 trips (it only yields 6 trips in total), so we finally conclude that the minimum time needed is `13`.

Following the binary search pattern, we get the minimum time required for all buses to collectively complete at least `totalTrips` trips which, in this example, is `13` units of time.

Solution Implementation

Python

```
from typing import List
from bisect import bisect_left

class Solution:
    def minimumTime(self, time: List[int], totalTrips: int) -> int:
        # Calculate the maximum possible time it would take to complete totalTrips.
        # This is done by taking the minimum time required for a single trip
        # and multiplying it by the total number of trips needed.
        max_time = min(time) * totalTrips

        # Use binary search to find the minimum amount of time needed
        # to complete the total number of trips.
        return bisect_left(
            range(max_time + 1),  # The search range includes 0 to max time
            totalTrips,           # The target value we are searching for
            # The key function calculates the total number of trips that
            # can be completed within 'current time' units of time for each worker.
            # It implements this by summing up how many trips each worker can complete.
            key=lambda current_time: sum(current_time // single_time for single_time in time)
        )
```

Java

```
class Solution {
    /**
     * Finds the minimum time required to complete the total number of trips.
     *
     * @param time      An array where each element represents the time it takes a bus to make one trip.
     * @param totalTrips The total number of trips that need to be completed.
     * @return The minimum time required to complete the total number of trips.
     */
    public long minimumTime(int[] time, int totalTrips) {
        // Initialize the variable to store the minimum time needed for one trip.
        int minTime = time[0];

        // Find the smallest trip time among all buses.
        for (int tripTime : time) {
            minTime = Math.min(minTime, tripTime);
        }

        // Set initial left and right bounds for binary search.
        long left = 1;
        long right = (long) minTime * totalTrips;

        // Perform binary search to find the minimum time required.
        while (left < right) {
            long mid = (left + right) >> 1; // Calculate the midpoint for the current range.
            long count = 0; // Initialize the count of trips that can be made by all buses in 'mid' time.

            // Calculate how many total trips can be made by all buses in 'mid' time.
            for (int tripTime : time) {
                count += mid / tripTime;
            }

            // If the count of trips is equal to or higher than required, search in the left half.
            if (count >= totalTrips) {
                right = mid;
            // Else search in the right half.
            } else {
                left = mid + 1;
            }
        }

        // The left pointer will point to the minimum time required when the search completes.
        return left;
    }
}
```

C++

```
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function to find the minimum time needed to make a number of trips.
    long long minimumTime(std::vector<int>& timePerTrip, int totalTrips) {
        // Find the minimum time per trip to estimate lower bound
        int minTime = *min_element(timePerTrip.begin(), timePerTrip.end());
        // Lower bound starts at 1 (minimum possible time)
        // Upper bound is the product of the minimum per trip and the total number of trips
        long long left = 1;
        long long right = static_cast<long long>(minTime) * totalTrips;

        // Perform binary search to find the minimum time needed
        while (left < right) {
            // Calculating the midpoint of our search interval
            long long mid = (left + right) / 2;
            long long tripsCompleted = 0;

            // Counting the total number of trips that can be completed in 'mid' time
            for (int time : timePerTrip) {
                tripsCompleted += mid / time;
            }

            // If the number of completed trips is at least the required amount,
            // we can try to find a smaller maximum time
            if (tripsCompleted >= totalTrips) {
                right = mid;
            // If not, we must increase the minimum time
            } else {
                left = mid + 1;
            }
        }
        // When the loop exits, 'left' will be our answer because 'right' will have converged to 'left'
        return left;
    }
};
```

TypeScript

```
function minimumTime(timePerTrip: number[], totalTrips: number): number {
    // Calculate the minimum time for a single trip to set the lower limit for binary search
    const minTime = Math.min(...timePerTrip);
    // Set initial lower bound as 1 since time cannot be 0
    let left: number = 1;
    // Set upper bound as minTime * totalTrips, assuming all trips at slowest speed
    let right: number = minTime * totalTrips;

    // Perform binary search to find the minimum time needed
    while (left < right) {
        // Find the mid-point time
        const mid: number = Math.floor((left + right) / 2);
        let tripsCompleted: number = 0;

        // Compute the number of trips that can be completed within 'mid' time
        for (const time of timePerTrip) {
            tripsCompleted += Math.floor(mid / time);
        }

        // If the calculated trips are equal or more than required, try to find a smaller time window
        if (tripsCompleted >= totalTrips) {
            right = mid;
        } else {
            // Otherwise, increase left to find a time period that allows completing the totalTrips
            left = mid + 1;
        }
    }

    // When the binary search is complete, 'left' will contain the minimum time needed
    return left;
}
```

```
from typing import List
from bisect import bisect_left

class Solution:
    def minimumTime(self, time: List[int], totalTrips: int) -> int:
        # Calculate the maximum possible time it would take to complete totalTrips.
        # This is done by taking the minimum time required for a single trip
        # and multiplying it by the total number of trips needed.
        max_time = min(time) * totalTrips

        # Use binary search to find the minimum amount of time needed
        # to complete the total number of trips.
        return bisect_left(
            range(max_time + 1),  # The search range includes 0 to max time
            totalTrips,           # The target value we are searching for
            # The key function calculates the total number of trips that
            # can be completed within 'current time' units of time for each worker.
            # It implements this by summing up how many trips each worker can complete.
            key=lambda current_time: sum(current_time // single_time for single_time in time)
        )
```

Time and Space Complexity

Time Complexity

The given code uses a binary search approach to find the minimum time required to make `totalTrips` using a list of `time` where each `time[i]` represents the time needed to complete one trip.

The time complexity of the binary search is $O(\log(max\_time))$, where `max_time = min(time) * totalTrips`. This represents the maximum amount of time it might take if the slowest worker alone had to complete all `totalTrips`.

Within each step of the binary search, the code executes a sum operation that runs in $O(n)$ time, where n is the number of workers (or the length of the `time` list). This sum operation calculates the total number of trips completed within a given time frame.

Therefore, the total time complexity of the code is $O(n * log(max\_time))$.

Space Complexity

The space complexity of the code is $O(1)$ since it only uses constant extra space for variables such as `mx` and the lambda function. There is no use of any additional data structures that grow with the input size.