2073. Time Needed to Buy Tickets

Easy Queue Array Simulation

Leetcode Link

Problem Description

In this problem, there are `n` individuals standing in a line to purchase tickets, with the queue arranged such that the `0th` person is at the start of the line and the `(n - 1)th` person is at the end of the line. An array `tickets` of length `n` is provided, where each element `tickets[i]` denotes the number of tickets the `i`th person wishes to buy.

The purchase process is such that each person requires exactly `1 second` to buy a single ticket, and can only buy one ticket at a time. After buying a ticket, if a person wants to buy more, they must go to the end of the line to purchase another ticket. This transition is assumed to be instantaneous. When an individual has no remaining tickets they wish to purchase, they leave the line.

The task is to calculate how much time it will take for the person in position `k` (using 0-based indexing) in the line to complete all their ticket purchases.

Intuition

To determine the total time required for the `kth` person to buy all their tickets, we have to simulate the ticket-buying process, respecting each person in the line. The main insights for the solution are:

- Every person in the line, including person `k`, will buy tickets in rounds. In each round, every person, starting from the front of the line, will buy one ticket if they still want to purchase tickets and then go to the end of the line.
- A person will leave the line as soon as they have no tickets left to buy.
- The person at the `kth` position will buy their last ticket and leave the line, and will not go to the end of the line again.

Based on these points, the solution iterates over each person in the queue:

- For people ahead of or at position `k`, the `kth` person will have to wait for each one of them to buy tickets up to the number of tickets person `k` wants (as both will get to buy their tickets). The minimum between `tickets[k]` (the number of tickets person `k` wants) and `tickets[i]` (the number of tickets the current person wants) is added to the total time.
- For people behind position `k`, the `kth` person will wait for each one of them to buy tickets only up to one less than the number of tickets they want (because after the `kth` person buys their last ticket, they will leave and not wait for these people anymore). Hence, the minimum between `tickets[k] - 1` (one less than the tickets `k` wants, because `k` won't buy a ticket after the final one) and `tickets[i]` is added to the total time.

The sum of all these minimums gives the total time taken for the person at position `k` to finish buying their tickets.

Solution Approach

The given solution follows a straightforward approach without the need for complex algorithms or data structures. The logic simply simulates the purchasing of tickets person by person. Here's a more detailed breakdown:

- Initialize `ans` to zero, which will accumulate the total time required for the `kth` person to buy their tickets.
- Loop through each person in the `tickets` array using `enumerate`, which provides both the index `i` and the number of tickets `t` that person wants.
  - For a person at or before the `kth` person in the queue (`i <= k`): Add the minimum of `tickets[k]` and `t` to `ans`. This represents the `kth` person waiting for each of the people in front of them, including themselves, to buy up to the same number of tickets that the `kth` person wants since they will all get to that number of tickets purchased before the `kth` person finishes.
  - For a person after the `kth` person in the queue (`i > k`): Add the minimum of `tickets[k] - 1` and `t` to `ans`. This represents the `kth` person waiting for each of the people behind them to buy their tickets, but only up until the point just before `k` buys their last ticket. Since the `kth` person won't queue again after buying their last ticket, they won't wait for any tickets bought by the people behind them this final time around.
- After the loop finishes, `ans` holds the total time taken for the person at position `k` to buy all their tickets, and that value is returned.

This solution ensures that the time the `kth` person will wait for others is correctly accounted for, adjusting for when they and the people in line ahead of them will buy tickets, and when they won't be in line anymore after buying their last ticket which is why they wait for one ticket less from those behind them. It's a clear and efficient way to simulate the queue process without the need for actual queue manipulation or any additional space, resulting in an O(n) time complexity where n is the number of people in the queue.

Example Walkthrough

Let's take a small example to illustrate the solution approach:

Suppose there are 5 individuals in a line (so `n = 5`) meaning to purchase tickets and their respective tickets are the number of tickets they want to buy represented by an array `tickets = [1, 2, 5, 2, 1]`. We want to find out how long it will take for the 3rd person (`k = 2`, 0-indexed) to buy all of their tickets.

Here is a step-by-step simulation following the solution approach:

- **Initialization:** `ans` starts at 0.
- **First Pass (i = 0, t = 1, k = 2):** The first person wants 1 ticket, which is less than what the 3rd person wants, so we add 1 to `ans`. `ans = 1`.
- **Second Pass (i = 1, t = 2, k = 2):** The second person wants 2 tickets, which is less than what the 3rd person wants, so we add 2 to `ans`. `ans = 1 + 2 = 3`.
- **Third Pass (i = 2, t = 5, k = 2):** The third person is the person in question, they want 5 tickets, so they will buy all their tickets. Add 5 to `ans`. `ans = 3 + 5 = 8`.
- **Fourth Pass (i = 3, t = 2, k = 2):** The fourth person wants 2 tickets. Since the 3rd person will not queue after buying their fifth ticket, we add min(5 - 1, 2) to `ans`, which is 2. `ans = 8 + 2 = 10`.
- **Fifth Pass (i = 4, t = 1, k = 2):** The fifth person wants 1 ticket, which is again less than what the 3rd person wants minus one, so we add 1 to `ans`. `ans = 10 + 1 = 11`.

After going through all individuals in the line:

- For individuals ahead of or at position `k` (the 3rd person), we added the minimum of the number of tickets they want and the number of tickets the 3rd person wants to `ans`.
- For individuals behind position `k`, we added the minimum of one less than the number of tickets the 3rd person wants and the number of tickets they want to `ans`.

The total time the 3rd person will wait is `ans = 11`. Hence, the 3rd person will take 11 seconds to buy all of their tickets.

Python Solution

```python
from typing import List

class Solution:
    def timeRequiredToBuy(self, tickets: List[int], k: int) -> int:
        # Initialize the total time required to 0
        total_time = 0

        # Iterate over the ticket queue to simulate the time passing
        for index, tickets_at_this_position in enumerate(tickets):
            # If the current position is before or at the target position k
            if index <= k:
                # Add the minimum of the target tickets and tickets at the current position
                # It ensures we do not count the extra tickets the target person doesn't need
                total_time += min(tickets[k], tickets_at_this_position)
            else:
                # After the target person has bought their tickets, they will not buy more
                # Thus, for the people after the target, we consider one less ticket for the target
                # Person at position k would have already bought their ticket when turn comes to later positions
                total_time += min(tickets[k] - 1, tickets_at_this_position)

        # Return the calculated total time
        return total_time

# Example usage:
# sol = Solution()
# print(sol.timeRequiredToBuy([2, 3, 2], 2))  # This would output 6, the total time to buy tickets
```

Java Solution

```java
class Solution {

    /**
     * Calculate the time required for a person to buy tickets.
     *
     * @param tickets Array representing the number of tickets each person in the queue needs.
     * @param k Index of the person whose time to buy tickets is being calculated.
     * @return The time required for the person at index k to buy their tickets.
     */
    public int timeRequiredToBuy(int[] tickets, int k) {
        // Initialize the answer variable to store the total time required.
        int totalTime = 0;

        // Loop through each person in the tickets array.
        for (int i = 0; i < tickets.length; i++) {
            // If the current person is before or at the position k in the queue,
            // they will buy tickets[i] amount of the same amount as the person
            // at the position k, whichever is smaller.
            if (i <= k) {
                totalTime += Math.min(tickets[k], tickets[i]);
            } else {
                // If the current person is after the position k in the queue,
                // they can only buy till tickets[k] - 1 as the person at k
                // will buy their last ticket before them.
                totalTime += Math.min(tickets[k] - 1, tickets[i]);
            }
        }

        // Return the total time required for the person at index k to buy their tickets.
        return totalTime;
    }
}
```

C++ Solution

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

class Solution {
public:
    // Calculates the time required to buy tickets
    int timeRequiredToBuy(vector<int>& tickets, int k) {
        int totalTime = 0; // Initialize the total time to 0
        int n = tickets.size(); // Get the number of people in the line

        for (int i = 0; i < n; ++i) {
            if (i <= k) {
                // If the person is before or at position k, only buy until the
                // number of tickets the kth person needs
                totalTime += min(tickets[k], tickets[i]);
            } else {
                // If the person is after the kth person, they can only buy until
                // one less than the number of tickets the kth person needs, because
                // when the kth gets their last ticket, these ones can't buy any more.
                totalTime += min(tickets[k] - 1, tickets[i]);
            }
        }
        return totalTime; // Return the total time calculated
    }
};
```

Typescript Solution

```typescript
function timeRequiredToBuy(tickets: number[], k: number): number {
    const totalPeople = tickets.length;
    let ticketsForTarget = tickets[k] - 1; // Subtract 1 because the target will purchase their last ticket in the end
    let totalTime = 0; // Initialize the total time needed

    // Round 1: Everyone purchases tickets up to the number of tickets the target person needs minus one
    for (let i = 0; i < totalPeople; i++) {
        if (tickets[i] <= ticketsForTarget) {
            totalTime += tickets[i]; // Add the number of tickets each person can buy without exceeding target
            tickets[i] = 0; // They have no more tickets to buy
        } else {
            totalTime += ticketsForTarget; // They can only buy as many as the target needs minus one
            tickets[i] -= ticketsForTarget; // Subtract the bought tickets from their total needed tickets
        }
    }

    // Round 2: Continue the purchasing process for remaining tickets
    // this time including the target person buying their last ticket
    for (let i = 0; i <= k; i++) {
        if (tickets[i] > 0) {
            totalTime += 1; // Add one extra time unit for every person including and before the target
        }
    }

    return totalTime;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the function is `O(n)`, where `n` is the length of the `tickets` array. This is because the function contains a single loop over the `tickets` array, and the operations within the loop are constant-time computations, involving simple arithmetic and comparison.

Space Complexity

The space complexity of the function is `O(1)` because it uses a fixed amount of extra space – the `ans` variable for storing the cumulative time. No additional data structures are used that grow with the input size.