1701. Average Waiting Time

Medium Array Simulation

Problem Description

In this problem, we are operating a restaurant with a single chef and a list of customers who arrive and place orders. Each customer has two properties:

- `arrival_i`: The arrival time for the `i-th` customer.
- `time_i`: The time required to prepare the order for the `i-th` customer.

Customers arrive in a non-decreasing order of their arrival times. The chef starts working on each order when he is not busy with another, and only works on one order at a time. If the customer arrives while the chef is busy, they need to wait until the chef can start their order. Once the chef starts working on an order, they work on it until it's finished before moving onto the next customer's order. Our task is to calculate the average waiting time of all customers. The waiting time for a customer consists of the time they wait before the chef begins their order and the time it takes to prepare their order.

Intuition

The key to solving this problem is to track the time at which each customer's order will be completed, and then calculate the total wait-time. To do this, we maintain a variable `t` that represents the current time when the chef finishes preparing the previous order. We iterate through all the customers, and for each customer, we update `t` to be the larger of `t` or the customer's arrival time (since the chef can only start preparing the order after the customer has arrived), then add the preparation time `b`. This gives us the time when the chef will complete the current order. For each customer, the waiting time is the difference between the time of completion `t` and the arrival time `a`. We sum up all these waiting times to get the total waiting time `tot`.

Solution Approach

To implement the given solution, we are following a straightforward approach using a simple loop without the need for complex data structures or algorithms. Here's a step-by-step breakdown of the solution:

1. Initialize `tot` to 0, which will hold the total waiting time for all customers.
2. Initialize `t` to 0, which will keep track of the time when the chef completes an order. Think of `t` as the chef's available time to start a new order.
3. Loop over each customer in the list of `customers`. For each customer denoted as `(a, b)`:
   - Update `t` to the maximum of `t` (the time when the chef will be free from the previous order) and `arrival_i` (the time when the current customer arrives). This is important because if a customer arrives before the chef is finished with the previous order, the chef can only start the next order after finishing the current one. However, if the chef is already free by the time the next customer arrives (`t < arrival_i`), they start the order at the customer's arrival time.
   - Add the preparation time `time_i` to `t`, i.e., `t = max(t, a) + b`. Now `t` represents the time when the order for the current customer will be completed.
   - Calculate the waiting time for the customer as the difference between the total time when their order is completed (`t`) and their arrival time (`a`), and add this to the total waiting time `tot`.
4. The average waiting time is computed by dividing `tot` by the number of customers `len(customers)`.
5. Return the average waiting time as a float, which corresponds to the problem requirement of calculating the average.

This solution works in O(n) time because it employs a single for-loop that goes through the customers, and O(1) extra space as it uses only two variables that keep track of the total waiting time and the current time, regardless of the number of customers.

Example Walkthrough

Suppose we have the following list of customers, where each pair represents the arrival time and the time it takes to prepare their order, respectively:

```
Customers = [(1, 2), (2, 5), (4, 3)]
```

Here's how the solution approach would be applied:

1. We initialize `tot` to 0. This will accumulate the total waiting time for all customers. We also initialize `t` to 0, representing when the chef can start the next order.
2. We then start iterating over the customers list:
   - For the first customer `(1, 2)`, since `t` (0) is less than `arrival_1` (1), we update `t` to `arrival_1` (1) and then add `time_1` (2), so `t` becomes 3. The waiting time for this customer is `t - arrival_1 = 3 - 1 = 2`, so we update `tot` to 2.
   - The second customer `(2, 5)` arrives when the chef is busy, so the chef can only start at time `t` (3). We update `t` to 3 and add `time_2` (5), which means `t` becomes 8. The waiting time for this customer is `8 - 2 = 6`, and we update `tot` to `2 + 6 = 8`.
   - The third customer `(4, 3)` arrives when the chef is free, as the chef finished the previous order at `t` (8), which is after the third customer's arrival. Therefore, `t` remains 8, and we add `time_3` (3), making `t` become 11. The waiting time for this customer is `11 - 4 = 7`, and the total waiting time `tot` is updated to `8 + 7 = 15`.
3. After iterating through all customers, we divide the total waiting time `tot` (15) by the number of customers (3), which gives us an average waiting time of `15 / 3 = 5.0`.
4. Finally, we return the average waiting time of 5.0 as the answer to the problem.

This example illustrates how the solution approach effectively calculates the average waiting time of customers in a restaurant using a simple loop and a straightforward update of the `tot` and `t` variables.

Solution Implementation

Python

```python
class Solution:
    def average_waiting_time(self, customers: List[List[int]]) -> float:
        # Initialize total waiting time and current time to zero.
        total_waiting_time = current_time = 0

        # Iterate over each customer.
        for arrival_time, service_time in customers:
            # If the current time is before the customer's arrival, wait until they arrive.
            # Otherwise, continue with the current time.
            current_time = max(current_time, arrival_time)

            # Add the service time to current time to service the customer.
            current_time += service_time

            # Calculate the waiting time for the current customer and add it to the total.
            waiting_time = current_time - arrival_time
            total_waiting_time += waiting_time

        # Calculate the average waiting time by dividing the total waiting time by the number of customers.
        average_waiting_time = total_waiting_time / len(customers)

        # Return the average waiting time.
        return average_waiting_time
```

Java

```java
class Solution {
    public double averageWaitingTime(int[][] customers) {
        double totalWaitingTime = 0; // Initialize total waiting time
        int currentTime = 0; // Initialize current time to track when the chef will be free

        // Iterate over each customer
        for (int[] customer : customers) {
            int arrivalTime = customer[0]; // Extract arrival time for the current customer
            int orderTime = customer[1];   // Extract order's cooking time for the current customer

            // Update current time: If the chef is free before the arrival, start at arrival time,
            // else start after finishing the last customer's order
            currentTime = Math.max(currentTime, arrivalTime) + orderTime;

            // Calculate waiting time for the current customer and add it to the total waiting time
            totalWaitingTime += currentTime - arrivalTime;
        }

        // Calculate the average waiting time by dividing the total waiting time by the number of customers
        return totalWaitingTime / customers.length;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // include algorithm for max

class Solution {
public:
    double averageWaitingTime(std::vector<std::vector<int>>& customers) {
        double totalWaitTime = 0; // Store the total waiting time for all customers.
        int currentTime = 0; // The current time to track when the chef finishes the orders.

        // Loop over each customer
        for (const auto& customer : customers) {
            int arrivalTime = customer[0]; // The time when the customer arrives.
            int orderTime = customer[1];   // The time taken to prepare the customer's order.

            // Update currentTime. If the chef is idle, set currentTime to the arrival time of the current customer.
            // Otherwise, add the order preparation time to the current time.
            currentTime = std::max(currentTime, arrivalTime) + orderTime;

            // The waiting time for the current customer is the total time since their arrival until the food is ready.
            // Add this to the total waiting time.
            totalWaitTime += currentTime - arrivalTime;
        }

        // Return the average waiting time, which is the total waiting time divided by the number of customers.
        return totalWaitTime / customers.size();
    }
};
```

TypeScript

```typescript
// A variable to store the total waiting time for all customers.
let totalWaitTime: number = 0;
// A variable to track the current time when the chef finishes the orders.
let currentTime: number = 0;

/**
 * Calculate the average waiting time for all customers.
 * @param customers An array of arrays, where each sub-array contains the arrival time and the order time for each customer.
 * @returns The average waiting time.
 */
function averageWaitingTime(customers: number[][]): number {
    // Loop through each customer in the array.
    for (const customer of customers) {
        // Extract the arrival and order time for the current customer.
        const arrivalTime: number = customer[0];
        const orderTime: number = customer[1];

        // Update currentTime: if the chef is idle (current time < arrival time),
        // set currentTime to the arrival time of the current customer. Otherwise, add the order
        // preparation time to the current time.
        currentTime = Math.max(currentTime, arrivalTime) + orderTime;

        // Calculate the waiting time for the current customer, which is the total time
        // from their arrival until the food is ready, and add it to the total wait time.
        totalWaitTime += currentTime - arrivalTime;
    }

    // Return the average waiting time by dividing the total waiting time by the number of customers.
    return totalWaitTime / customers.length;
}
```

```python
class Solution:
    def average_waiting_time(self, customers: List[List[int]]) -> float:
        # Initialize total waiting time and current time to zero.
        total_waiting_time = current_time = 0

        # Iterate over each customer.
        for arrival_time, service_time in customers:
            # If the current time is before the customer's arrival, wait until they arrive.
            # Otherwise, continue with the current time.
            current_time = max(current_time, arrival_time)

            # Add the service time to current time to service the customer.
            current_time += service_time

            # Calculate the waiting time for the current customer and add it to the total.
            waiting_time = current_time - arrival_time
            total_waiting_time += waiting_time

        # Calculate the average waiting time by dividing the total waiting time by the number of customers.
        average_waiting_time = total_waiting_time / len(customers)

        # Return the average waiting time.
        return average_waiting_time
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the input list `customers`. This is because the code involves a single loop that iterates through each customer exactly once, performing a constant amount of work for each customer without any nested loops.

In the loop, two major operations are performed for each customer: calculating the time at which the chef starts preparing the customer's food (`max(t, a) + b`) and updating the total waiting time (`tot += t - a`). Since both of these operations are executed in constant time, the time complexity remains linear with respect to the number of customers.

Space Complexity

The space complexity of the given code is `O(1)`. Aside from the input list `customers`, only a fixed number of integer variables (`tot` and `t`) are used for calculations. These variables do not depend on the size of the input and, as such, do not scale with the input. Regardless of the number of customers, the space used by the algorithm is constant.

Thus, the space required for the algorithm does not grow with the size of the input, which results in constant space complexity.