2224. Minimum Number of Operations to Convert Time

Easy Greedy String

Problem Description

The problem presents a scenario where you are given two strings, `current` and `correct`, which represent two 24-hour formatted times, using the format "HH:MM". The `HH` part corresponds to the hour portion, ranging from 00 to 23, and `MM` is the minutes portion, ranging from 00 to 59. The goal is to find the minimum number of operations required to change the `current` time to the `correct` time. An operation is defined as adding 1, 5, 15, or 60 minutes to the `current` time, and you are allowed to perform as many operations as necessary to achieve the `correct` time.

Intuition

To solve this problem, we need to minimize the number of operations required to convert `current` to `correct`. The intuition is to use the largest possible increment (60 minutes) as many times as we can without exceeding the `correct` time, then proceed to the next largest (15 minutes), and so on down to the smallest increment (1 minute). This greedy approach ensures that we are always making the biggest leap towards the `correct` time at each step, thereby minimizing the total number of steps.

We convert both `current` and `correct` times to minutes since midnight, which makes the calculation easier since we are now dealing with integers. This is done by multiplying the hours by 60 and adding the minutes. Then, we calculate the difference `d` between `correct` and `current`.

Next, we use a loop to iterate over the list of increments `[60, 15, 5, 1]`. In each iteration, we add to `ans` the integer division of `d` by the current increment, which is the maximum number of times we can perform the current operation. We then update `d` to the remainder of this division to process the remaining time with the next smaller increment. This continues until we have used all permissible increments, at which point `d` should be 0, and `ans` should contain the minimum number of operations.

Solution Approach

The implementation of the solution closely mirrors the intuition explained earlier. The algorithm is straightforward and can be broken down into the following steps:

1. **Convert both `current` and `correct` into minutes:** This is the first operation in the solution which involves parsing the hour and minute components of both time strings separately, converting them to integers, multiplying the hours by 60 to get the total minutes for the hours part, and finally adding the minutes to this result. In Python, this looks like `int(current[:2]) * 60 + int(current[3:])`, where `[:2]` takes the first two characters of the string (hours) and `[3:]` takes the characters from the fourth to the end (minutes).
2. **Calculate the difference:** We then subtract the value of `current` in minutes from the value of `correct` in minutes to get the total number of minutes that need to be added to `current` to reach `correct`. This difference is represented by `d`.
3. **Perform operations to minimize the time difference:** Using a `for` loop, the solution iterates over a list `[60, 15, 5, 1]` which contains the allowed minute increments for the operations, in descending order.
   - Within the loop, the solution uses integer division `// 1` to find out how many times a particular increment can be used. It adds this number to the `ans` variable, which accumulates the total number of operations needed.
   - Then, `d` is updated to the remainder of this division with `d %= 1`. This remainder represents the minutes that are yet to be matched and is efficiently reduced with each larger increment before moving to smaller ones.

The choice of data structures in this solution is minimal. An integer `ans` is used to store the cumulative count of operations, and a simple list of increments is iterated upon.

This approach is a common algorithmic pattern called the greedy algorithm, which tries to solve a problem by making the best possible decision at the current step without looking ahead to the future steps. It's called "greedy" because the decision is based on what seems to be the best immediate choice.

Lastly, the algorithm executes in O(1) time complexity since the operations are bound by a fixed number of possible increment values and the loop runs at most 4 times (once for each increment value: 60, 15, 5, and 1).

Example Walkthrough

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

Suppose we have `current` time as `"02:30"` and `correct` time as `"03:10"`.

1. **Convert both times to minutes**:
   - The `current` time is converted to 150 minutes (`2*60 + 30`).
   - The `correct` time is converted to 190 minutes (`3*60 + 10`).
2. **Calculate the difference**:
   - The difference `d` between `correct` and `current` is 40 minutes (`190 - 150`).
3. **Perform operations**:
   - We start by iterating over the increments `[60, 15, 5, 1]`.
   - We cannot use the 60-minute operation since `d` is less than 60.
   - For the 15-minute operation, we add `40 // 15 = 2`, so we can add 15 minutes two times. `ans` becomes `2`, and `d` becomes `10` (the remainder of dividing 40 by 15).
   - The 5-minute operation can be used twice on the remaining 10 minutes. Now, `ans` is `2 + 2 = 4`, and `d` is `0`.
   - Since `d` is now 0, we do not need to perform any 1-minute operations.

In this example, the minimum number of operations required to change the `current` time to the `correct` time is 4. This includes performing the 15-minute increment twice and the 5-minute increment twice.

Solution Implementation

Python

```python
class Solution:
    def convertTime(self, current: str, correct: str) -> int:
        # Convert the 'current' time to minutes
        current_minutes = int(current[:2]) * 60 + int(current[3:])

        # Convert the 'correct' time to minutes
        correct_minutes = int(correct[:2]) * 60 + int(correct[3:])

        # Calculate the difference in minutes
        delta_minutes = correct_minutes - current_minutes

        # Initialize the number of operations used
        operations_used = 0

        # List of available operation increments in minutes
        operation_increments = [60, 15, 5, 1]

        # For each increment option, calculate the maximum number of operations possible
        # And then, reduce the remaining delta_minutes accordingly
        for increment in operation_increments:
            operations_used += delta_minutes // increment  # Perform the largest operations possible
            delta_minutes %= increment  # Reduce delta_minutes by the amount operated on

        # Return the total number of operations used to adjust the time
        return operations_used
```

Java

```java
class Solution {
    public int convertTime(String current, String correct) {
        // Convert the 'current' time into minutes
        int currentMinutes = Integer.parseInt(current.substring(0, 2)) * 60
                           + Integer.parseInt(current.substring(3));

        // Convert the 'correct' time into minutes
        int correctMinutes = Integer.parseInt(correct.substring(0, 2)) * 60
                           + Integer.parseInt(correct.substring(3));

        // Initialize answer to store the number of operations required
        int operations = 0;

        // Calculate the difference in minutes
        int difference = correctMinutes - currentMinutes;

        // Array containing possible operations in descending order by their time value
        Integer[] operationsArray = new Integer[] {60, 15, 5, 1};

        // Iterate over the operations array to find the minimum number
        // of operations needed to reach the correct time
        for (int minutesPerOperation : operationsArray) {
            // Divide the time difference by the time value of operation
            operations += difference / minutesPerOperation;

            // Update the difference to reflect the remaining time after performing this operation
            difference %= minutesPerOperation;
        }

        // Return the minimum number of operations required
        return operations;
    }
}
```

C++

```cpp
#include <string>
#include <vector>

class Solution {
public:
    // Function to convert time from 'current' to 'correct' using minimum operations
    int convertTime(std::string current, std::string correct) {
        // Convert 'current' time from hours:minutes format to minutes
        int currentMinutes = std::stoi(current.substr(0, 2)) * 60 + std::stoi(current.substr(3, 2));
        // Convert 'correct' time from hours:minutes format to minutes
        int correctMinutes = std::stoi(correct.substr(0, 2)) * 60 + std::stoi(correct.substr(3, 2));

        // Calculate the difference in minutes
        int difference = correctMinutes - currentMinutes;
        // Variable to store the number of operations needed
        int operationsCount = 0;
        // Define increments in minutes that can be used to adjust the time
        std::vector<int> increments = {60, 15, 5, 1};

        // Iterate over possible increments
        for (int increment : increments) {
            // Use the largest increment to reduce the difference as much as possible
            operationsCount += difference / increment;
            // Update the remaining difference
            difference %= increment;
        }
        // Return the total number of operations required
        return operationsCount;
    }
};
```

TypeScript

```typescript
// Import statements are not typically used in TypeScript as they are in C++
// Instead, TypeScript uses modules and export/import syntax

// Function to convert time from 'current' to 'correct' using minimum operations
function convertTime(current: string, correct: string): number {
    // Convert 'current' time from hours:minutes format to minutes
    const currentMinutes: number = parseInt(current.substring(0, 2)) * 60 + parseInt(current.substring(3));

    // Convert 'correct' time from hours:minutes format to minutes
    const correctMinutes: number = parseInt(correct.substring(0, 2)) * 60 + parseInt(correct.substring(3));

    // Calculate the difference in minutes
    let difference: number = correctMinutes - currentMinutes;

    // Variable to store the number of operations needed
    let operationsCount: number = 0;

    // Define increments in minutes that can be used to adjust the time
    const increments: number[] = [60, 15, 5, 1];

    // Iterate over possible increments
    for (const increment of increments) {
        // Use the largest increment to reduce the difference as much as possible
        operationsCount += Math.floor(difference / increment);

        // Update the remaining difference
        difference %= increment;
    }

    // Return the total number of operations required
    return operationsCount;
}

// The code as per TypeScript conventions does not define global variables or methods.
// Instead, they are typically encapsulated in modules or classes.
// However, as per the request, this function is defined globally.
```

```python
class Solution:
    def convertTime(self, current: str, correct: str) -> int:
        # Convert the 'current' time to minutes
        current_minutes = int(current[:2]) * 60 + int(current[3:])

        # Convert the 'correct' time to minutes
        correct_minutes = int(correct[:2]) * 60 + int(correct[3:])

        # Calculate the difference in minutes
        delta_minutes = correct_minutes - current_minutes

        # Initialize the number of operations used
        operations_used = 0

        # List of available operation increments in minutes
        operation_increments = [60, 15, 5, 1]

        # For each increment option, calculate the maximum number of operations possible
        # And then, reduce the remaining delta_minutes accordingly
        for increment in operation_increments:
            operations_used += delta_minutes // increment  # Perform the largest operations possible
            delta_minutes %= increment  # Reduce delta_minutes by the amount operated on

        # Return the total number of operations used to adjust the time
        return operations_used
```

Time and Space Complexity

Time Complexity

The time complexity of this function is `O(1)`. This is because the function performs a constant number of operations irrespective of the size of the input. The calculations performed to convert the times from string format to minutes, the loop to calculate the number of operations needed to reach the correct time, and the arithmetic operations inside the loop all execute in constant time. There are no iterative statements that depend on the input size, and the loop runs a maximum of four times (once for each value in the `[60, 15, 5, 1]` list).

Space Complexity

The space complexity of this function is also `O(1)`. The function only uses a fixed amount of additional memory: the variables `a`, `b`, `ans`, and `d` hold single integer values, and the list `[60, 15, 5, 1]` has a fixed size. Therefore, the amount of memory used does not scale with the input size, guaranteeing constant space complexity.