

# 2651. Calculate Delayed Arrival Time

EasyMath

## Problem Description

In this problem, you are given two integers representing time in a 24-hour format. The first integer, `arrivalTime`, denotes the scheduled arrival time of a train at a station. The second integer, `delayedTime`, corresponds to the number of hours the train is delayed. Your task is to calculate the new arrival time of the train after accounting for the delay. The challenge here is to correctly handle the day's transition when the delay pushes the arrival time past midnight. Since time is given in a 24-hour format, it means that the hours reset after reaching 23, with the next hour being 00, which signifies the start of a new day. The "return the time when the train will arrive at the station" implies calculating the sum of `arrivalTime` and `delayedTime` and adjusting it if necessary to fit within the 24-hour clock constraints.

## Intuition

To approach this problem, we simply need to calculate the new arrival time by adding `arrivalTime` and `delayedTime` together. The key point is that we need to wrap around the clock if the summation exceeds the 24-hour format limit (which is 23 hours). This scenario is similar to how an analog clock behaves, as the hands continue to rotate and start over after completing a full cycle (12 hours for an analog clock, and 24 hours for a digital clock in this case).

To simulate this behavior, we can use the modulo operation. The modulo operation, denoted by `%`, returns the remainder of the division of two numbers. In this case, we can apply the modulo operation with the divisor being 24 to ensure that the result fits within the 24-hour format. If the result is less than 24 (which implies no day transition), the modulo operation won't affect the calculation, and the return value is the same as the direct sum. If the result is more than or equal to 24 (indicating the next day), the modulo operation will give us the correct adjusted time.

Therefore, the intuition behind the solution is quite straightforward. We take `arrivalTime` and `delayedTime`, perform an addition to find the delayed arrival time, and then apply the modulo operation with 24 to wrap around and get the correct time within the 24-hour clock system. The result of this calculation is the actual time of arrival after incorporating the delay.

## Solution Approach

In the given solution, the implementation requires no complex algorithms, additional data structures, or intricate patterns. It takes advantage of a simple, yet powerful arithmetic property of numbers — namely, the modulo operation.

The solution approach can be broken down into the following steps:

- Calculate the total time by summing the `arrivalTime` and `delayedTime`.
- Apply modulo operation with 24 to the result from step 1 to ensure the time is wrapped around the 24-hour format.

The sole line of code in the given Python solution captures the entire implementation:

```
return (arrivalTime + delayedTime) % 24
```

This line of code performs the addition of `arrivalTime` and `delayedTime` to calculate the total hours. The use of the modulo operation with 24 subsequently ensures that if the addition exceeds 23 (the maximum hour value in a day), we cycle back to 0, thus mimicking the behavior of a 24-hour clock.

Let's consider an example where `arrivalTime` is 22 hours (or 10 PM) and `delayedTime` is 4 hours:

- Without applying the modulo operation, the sum would be 26, which is not a valid time in a 24-hour format.
- With the modulo operation (  $(22 + 4) \% 24$  ), we get  $26 \% 24$ , which equals 2. This correctly represents the new arrival time as 2 AM the following day.

The Python `%` operator is all that's required for this calculation, as it inherently implements this 'wrap-around' behavior by giving us the remainder from the division. No other algorithms or data structures are necessary since the problem deals strictly with integer arithmetic and leverages the inherent properties of numbers and the modulo operation.

## Example Walkthrough

Let's take a small example to illustrate how the solution works. Consider that the `arrivalTime` of a train is 18 hours (6 PM) and the `delayedTime` is 8 hours.

Following the solution approach:

- Calculate the total time by summing the `arrivalTime` and `delayedTime`. In our example:

```
totalTime = arrivalTime + delayedTime
totalTime = 18 + 8
totalTime = 26
```

- Apply the modulo operation with 24 to the result from step 1 to fit it within the 24-hour format.

```
newArrivalTime = totalTime % 24
newArrivalTime = 26 % 24
newArrivalTime = 2
```

By using the modulo operation (`% 24`), the sum of 26 is adjusted back within the range of the 24-hour system. Instead of an invalid hour of 26, we get a valid result of 2. The train was supposed to arrive at 6 PM but due to the 8-hour delay, it will now arrive at 2 AM the next day.

The final Python code carries out this simple calculation and returns `2` as the new arrival time, properly formatted within the 24-hour clock system.

## Solution Implementation

### Python

```
class Solution:
    def findDelayedArrivalTime(self, arrival_time: int, delay_time: int) -> int:
        # Adds the delay time to the arrival time
        # Then uses modulo operation to adjust for the 24-hour clock format
        return (arrival_time + delay_time) % 24
```

### Java

```
class Solution {
    // Computes the new arrival time after a delay
    // @param arrivalTime is the scheduled arrival time
    // @param delayedTime is the delay duration
    // @return the new arrival time after accounting for the delay
    public int findDelayedArrivalTime(int arrivalTime, int delayedTime) {
        // Calculate the delayed arrival time by adding the delay to the original arrival time
        // Use modulo operation to wrap around the clock if the time exceeds 24 hours
        int newArrivalTime = (arrivalTime + delayedTime) % 24;

        // Return the calculated new arrival time
        return newArrivalTime;
    }
}
```

### C++

```
class Solution {
public:
    // Function to calculate the delayed arrival time
    int findDelayedArrivalTime(int arrivalTime, int delayedTime) {
        // The sum of arrivalTime and delayedTime can exceed 24 hours,
        // so we use modulo operation to get the time within a 24-hour format
        int delayedArrivalTime = (arrivalTime + delayedTime) % 24;

        return delayedArrivalTime;
    }
};
```

### TypeScript

```
// This function calculates the new arrival time after a delay
// `arrivalTime` is the scheduled time of arrival (0-23 hours)
// `delayedTime` is the time by which the arrival is delayed
// Returns the updated arrival time in hours (mod 24 format)
function findDelayedArrivalTime(arrivalTime: number, delayedTime: number): number {
    // Ensure the arrival time is within a 24-hour format after adding the delay
    const newArrivalTime = (arrivalTime + delayedTime) % 24;

    // Return the updated arrival time
    return newArrivalTime;
}
```

```
class Solution:
    def findDelayedArrivalTime(self, arrival_time: int, delay_time: int) -> int:
        # Adds the delay time to the arrival time
        # Then uses modulo operation to adjust for the 24-hour clock format
        return (arrival_time + delay_time) % 24
```

## Time and Space Complexity

The time complexity of the given code is  $O(1)$ , meaning it takes a constant amount of time to compute the result, regardless of the size of the input. This is because the code performs a fixed number of operations: one addition and one modulo operation.

The space complexity is also  $O(1)$ , indicating that a constant amount of space is used regardless of the input size. The function only calculates a single integer result and does not use any additional memory that scales with the input size.