2446. Determine if Two Events Have Conflict

Easy Array String Leetcode Link

Problem Description

The problem presents us with a scenario where two events are happening on the same day and we want to determine if they conflict with each other. Each event provides a start time and an end time in 24-hour format (`HH:MM`). A conflict is defined as the presence of a common time interval between the two events, even if it's just a minute. For example, if `event1` ends at `12:00` and `event2` starts at `12:00`, there is no conflict, but if `event2` starts even one minute before `event1` ends, there is a conflict. We are required to return `true` if there's a conflict and `false` otherwise.

Intuition

To determine if there is a conflict, we need to compare the time intervals of the two events. There are a few scenarios where no conflict can exist, which are important to identify.

1. The first event ends before the second event starts. In this case, `endTime1` is less than `startTime2`, so there is no overlap.
2. The second event ends before the first event starts. Here, `endTime2` is less than `startTime1`, and again, there is no overlap.

The solution cuts straight to the heart of these scenarios. It performs a simple logical check to see if the two time periods don't overlap. If the start time of `event1` is after the end time of `event2` or the end time of `event1` is before the start time of `event2`, then the two events can't possibly conflict; they are entirely separate in time. This can be written concisely as:

```
not (event1[0] > event2[1] or event1[1] < event2[0])
```

This expression evaluates to `True` if there is a conflict (event times overlap) and `False` if there is no conflict. The `not` operator negates the condition of no overlap, giving us the correct condition for detecting a conflict.

The elegance of the solution lies in its simplicity and the fact that it covers all possible scenarios where a conflict could arise with only one line of logical comparison.

Solution Approach

The solution to the problem uses a simple comparison-based approach, which falls under the category of interval scheduling algorithms. Interval scheduling problems typically involve finding optimum schedules among a set of events based on their start and end times. However, in this scenario, the task is not to schedule but to determine if there is an overlap between the two given intervals.

The data structures used in the solution are minimal—it simply takes the input in the form of two lists, `event1` and `event2`, which consist of string elements representing the start and end times of the events.

The code uses a single Boolean expression to identify the condition where no conflict occurs. Here's a walkthrough of that logic:

```python
class Solution:
    def haveConflict(self, event1: List[str], event2: List[str]) -> bool:
        # Return False if event1 ends before event2 starts or vice versa
        return not (event1[0] > event2[1] or event1[1] < event2[0])
```

Let's dissect the code:

- `event1[0] > event2[1]`: This comparison checks if the start time of `event1` is greater than the end time of `event2`. If this is true, it means `event1` starts *after* `event2` has ended, thus no conflict exists.
- `event1[1] < event2[0]`: Conversely, this comparison checks if the end time of `event1` is lesser than the start time of `event2`. This condition suggests that `event1` ends *before* `event2` starts, also indicating no conflict.

The `or` operator is used to combine these two comparisons. If either condition is true, there is no overlap; thus, `event1` and `event2` do not conflict.

The `not` operator at the beginning negates the result of the combined condition. If there is no overlap, the `not` operator turns the result to `False`, indicating no conflict. If there is an overlap (meaning both comparisons are false), the `not` operation turns the result to `True`, signaling a conflict.

There are no special algorithms or advanced data structures necessary for this solution, thanks to the simplicity of the problem. It relies on basic comparisons and logical operators to evaluate the conditions efficiently.

Example Walkthrough

Let's consider a small example to illustrate the solution approach.

Suppose we have two events:

- `event1` starts at `09:30` and ends at `11:00`
- `event2` starts at `10:45` and ends at `12:30`

Using the 24-hour format, here's how the events look:

- `event1` is represented as `['09:30', '11:00']`
- `event2` is represented as `['10:45', '12:30']`

To determine if there is a conflict between `event1` and `event2`, we'll use the following logic from the solution approach:

```python
def haveConflict(event1: List[str], event2: List[str]) -> bool:
    return not (event1[0] > event2[1] or event1[1] < event2[0])
```

- First, we check if `event1` starts after `event2` ends; in our case, does `'09:30' > '12:30'?`. This is not true.
- Second, we check if `event1` ends before `event2` starts; does `'11:00' < '10:45'?`. This is also not true.

Since neither of the conditions for no conflict is met (because both comparisons returned false), the combined result of the comparisons within the parentheses is `False`.

By applying the `not` operator, the resultant value becomes `True`, indicating that there is a conflict. In this case, `event1` ends at `11:00`, which is after `event2` has started at `10:45`, thus they share a time overlap from `10:45` to `11:00`.

Therefore, when we input the given times into the function:

```python
event1 = ['09:30', '11:00']
event2 = ['10:45', '12:30']

print(haveConflict(event1, event2)) # Output: True
```

The output is `True`, reflecting that there is a conflict between these two events.

Python Solution

```python
from typing import List

class Solution:
    def have_conflict(self, event1: List[str], event2: List[str]) -> bool:
        """
        Check if two events conflict based on their start and end times.

        Args:
        event1 (List[str]): The Start and end time of the first event [start, end]
        event2 (List[str]): The Start and end time of the second event [start, end]

        Returns:
        bool: True if there is a conflict between the events, False otherwise.
        """
        # Convert the start and end times to a comparable format if necessary
        # Assuming event1 and event2 are in correct format and no conversion is needed.

        # The events do not conflict if event1 ends before event2 starts
        # or if event1 starts after event2 ends. Otherwise, they do conflict.
        return not (event1[1] < event2[0] or event1[0] > event2[1])

# Example Usage
sol = Solution()
# Assuming the times are in 'HH:MM' format and events are ['start time', 'end time']
print(sol.have_conflict(["09:00", "10:00"], ["10:01", "11:00"])) # Should print False, no conflict
print(sol.have_conflict(["09:00", "10:00"], ["09:30", "10:30"])) # Should print True, there is a conflict
```

Java Solution

```java
class Solution {

    /**
     * Check if two events have a conflict based on their start and end times.
     *
     * @param event1 Array with 2 strings representing the start and end times of the first event.
     * @param event2 Array with 2 strings representing the start and end times of the second event.
     * @return boolean value that indicates whether the events conflict.
     */
    public boolean haveConflict(String[] event1, String[] event2) {
        // Event 1 ends before Event 2 starts -> No conflict
        boolean event1EndsBeforeEvent2Starts = event1[1].compareTo(event2[0]) < 0;

        // Event 1 starts after Event 2 ends -> No conflict
        boolean event1StartsAfterEvent2Ends = event1[0].compareTo(event2[1]) > 0;

        // If either of the above conditions is true,
        // there is no conflict, so we return the negation of any conflict.
        return !event1EndsBeforeEvent2Starts && !event1StartsAfterEvent2Ends;
    }
}
```

C++ Solution

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

class Solution {
public:
    // Function to check if two events have a conflict based on their time ranges
    bool haveConflict(std::vector<std::string>& event1, std::vector<std::string>& event2) {
        // event1: [start_time1, end_time1]
        // event2: [start_time2, end_time2]

        // Convert event's start and end times to comparable strings
        std::string start_time1 = event1[0];
        std::string end_time1 = event1[1];
        std::string start_time2 = event2[0];
        std::string end_time2 = event2[1];

        // Compare the times to determine if there is a conflict
        // No conflict if event1 ends before event2 starts or event1 starts after event2 ends
        return !(end_time1 < start_time2 || start_time1 > end_time2);
    }
};
```

Typescript Solution

```typescript
// Function to determine if two events have a conflict based on their time intervals.
// @param event1 The time interval of the first event as an array with start and end times.
// @param event2 The time interval of the second event as an array with start and end times.
// @returns A boolean value indicating whether there is a conflict (true) or not (false).
function haveConflict(event1: string[], event2: string[]): boolean {
    // Check if the first event ends before the second event starts or
    // the first event starts after the second event ends.
    // If either condition is true, there is no conflict.
    const event1StartsAfterEvent2Ends: boolean = event1[0] > event2[1];
    const event1EndsBeforeEvent2Starts: boolean = event1[1] < event2[0];

    // Return the inverse of either condition.
    // If both conditions are false (meaning there is overlap), the result is true (conflict exists).
    return !(event1StartsAfterEvent2Ends || event1EndsBeforeEvent2Starts);
}
```

Time and Space Complexity

Time Complexity

The time complexity of the function `haveConflict` is `O(1)`. This is because the function performs a constant number of operations regardless of the input size. It compares the start and end times of two events, which involves just a few comparisons and logical operations.

Space Complexity

The space complexity of the function `haveConflict` is also `O(1)`. The function does not use any additional data structures that grow with the input size. It only uses a fixed amount of space to store the input parameters and perform the comparison operations.