# 2409. Count Days Spent Together

Easy Math String Leetcode Link

## Problem Description

Alice and Bob are both visiting Rome on separate business trips, and we're given specific dates for when they will arrive and when they will leave. The objective is to calculate the total number of days both Alice and Bob will be in Rome simultaneously. We are provided with four dates: `arriveAlice` and `leaveAlice` for Alice's visit, and `arriveBob` and `leaveBob` for Bob's visit. Each date is given as a string in the "MM-DD" format, where "MM" represents the month and "DD" represents the day.

The problem specifies that the time period we're considering occurs within the same year, and importantly, it is not a leap year, so February has 28 days. The number of days per month can be represented as `[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]` corresponding to January through December, respectively.

We need to return the number of days that Alice and Bob will be in the city at the same time. This is inclusive of both their arrival and departure dates if they overlap at all.

## Intuition

The intuition behind the provided solution is to find the common range of dates when both Alice and Bob are in Rome and then calculate the length of this overlap. The approach includes the following steps:

1. Determine the latest arrival date between Alice and Bob. This is done by comparing `arriveAlice` and `arriveBob` and taking the maximum of the two. This marks the start of the period when both are in Rome.

2. Determine the earliest leave date between Alice and Bob by comparing `leaveAlice` and `leaveBob` and taking the minimum of the two. This marks the end of the period when both are in Rome.

3. With these boundaries (start and end), calculate the total number of days they overlap. This requires converting the dates into a cumulative day count from the start of the year and then finding the difference between the start and end dates.

4. The day count for each date is found by summing the total number of days in the months leading up to the given month using the provided array for the days in each month, then adding the day of the month from the date.

5. Once we have the day counts for both the start and end of the overlap, we calculate the difference. If the latest arrival is after the earliest departure, this would result in a negative number, indicating no overlap. Since we cannot have negative days of overlap, we use the `max` function to return zero in such cases.

6. If there is an overlap, we add 1 to the difference since both the start and end dates are inclusive.

By using these steps, we efficiently find the number of days two intervals overlap, which corresponds to the number of days Alice and Bob are in Rome simultaneously.

## Solution Approach

The implementation of the solution uses straightforward calculations and built-in Python functions to compute the number of overlapping days. Below are the specific steps detailed with the algorithms and data structures used:

1. **Determine Overlap Start and End:**
   - The solution defines two variables, `a` and `b`, using the built-in `max` and `min` functions to calculate the latest arrival date and the earliest departure date, respectively.
   - `a` and `b` are used to represent the start and end of the overlapping period.
2. **Prepare Data Structure for Days per Month:**
   - A tuple named `days` is used to store the number of days in each month for a non-leap year. This is a fixed-size, immutable data structure, which is appropriate for the constant values representing the days in each month.
3. **Convert Dates to Day Counts:**
   - To convert the dates in "MM-DD" format to a cumulative count of days from the start of the year, the solution uses string slicing and the `sum` function.
   - For example, `sum(days[: int(a[:2]) - 1])` calculates the total days up to the start of the month represented by `a`. It slices the `days` tuple up to the month index (subtracting 1 since Python uses 0-indexing) and sums the values.
   - Then, it adds the day of the month from `a` (or `b` for the end date): `int(a[3:])`.
4. **Calculate Overlapping Days:**
   - With the day counts for the start (`x`) and end (`y`) of the overlap determined, the difference `y - x` gives the number of days in between these dates, not including the start date.
   - Since the problem states the dates are inclusive, 1 is added to the difference.
   - To ensure there is no negative count of overlap (which would represent no overlap at all), the `max` function is used to return either the calculated overlap or `0` if the overlap calculation is negative: `max(y - x + 1, 0)`.

By working through these steps, the solution effectively transforms the date range comparison problem into one of simple arithmetic and takes advantage of Python's powerful built-in functions and data structures to do so efficiently.

### Example Walkthrough

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

Assume we have the following travel dates for Alice and Bob:

- Alice's travel dates: `arriveAlice = "02-08"` and `leaveAlice = "02-15"`
- Bob's travel dates: `arriveBob = "02-09"` and `leaveBob = "02-18"`

Following the solution approach:

1. **Determine Overlap Start and End:**
   - Compare Alice's and Bob's arrival dates, find the latest arrival date: `arriveBob` is "02-09", which is later than `arriveAlice`.
   - Compare Alice's and Bob's leave dates, find the earliest leave date: `leaveAlice` is "02-15", which is earlier than `leaveBob`.
   - Thus, the overlap starts on "02-09" and ends on "02-15".
2. **Prepare Data Structure for Days per Month:**
   - We already have a tuple that represents the days in each month: `(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)`.
3. **Convert Dates to Day Counts:**
   - To calculate the day count for "02-09" (`arriveBob`), we sum the days for January and add the days in February up to the 9th: `31 + 9 = 40`.
   - To calculate the day count for "02-15" (`leaveAlice`), we sum the days for January and add the days in February up to the 15th: `31 + 15 = 46`.
4. **Calculate Overlapping Days:**
   - The difference between the day counts is `46 - 40 = 6`. Since both the start and end dates are inclusive, we add 1 to this difference, giving us `6 + 1 = 7`.
   - The overlap is 7 days, meaning Alice and Bob will be in Rome simultaneously for a period of 7 days.

By executing these steps with the given dates and applying the logic and calculations described in the solution approach, we find that Alice and Bob will be in Rome at the same time for a total of 7 days.

## Python Solution

```python
class Solution:
    def countDaysTogether(self, arrive_alice: str, leave_alice: str, arrive_bob: str, leave_bob: str) -> int:
        # Retrieve the later arrival date between Alice and Bob
        later_arrival = max(arrive_alice, arrive_bob)

        # Retrieve the earlier departure date between Alice and Bob
        earlier_departure = min(leave_alice, leave_bob)

        # Tuple containing the number of days in each month
        days_in_months = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

        # Calculate the day of the year for the later arrival date
        arrival_day_of_year = sum(days_in_months[:int(later_arrival[:2]) - 1]) + int(later_arrival[3:])

        # Calculate the day of the year for the earlier departure date
        departure_day_of_year = sum(days_in_months[:int(earlier_departure[:2]) - 1]) + int(earlier_departure[3:])

        # Calculate and return the number of shared days between Alice and Bob
        # Add 1 because if they arrive and leave on the same day, they have spent 1 day together
        shared_days = max(departure_day_of_year - arrival_day_of_year + 1, 0)
        return shared_days

```

## Java Solution

```java
class Solution {

    // Array to store the number of days in each month considering a non-leap year.
    private int[] daysInMonth = new int[] {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

    /**
     * Calculates the number of days Alice and Bob spend together.
     *
     * @param arriveAlice Arrival date of Alice in "MM-DD" format.
     * @param leaveAlice  Departure date of Alice in "MM-DD" format.
     * @param arriveBob   Arrival date of Bob in "MM-DD" format.
     * @param leaveBob    Departure date of Bob in "MM-DD" format.
     * @return The number of days they spend together.
     */
    public int countDaysTogether(
        String arriveAlice, String leaveAlice, String arriveBob, String leaveBob) {

        // Determine the later arrival date of the two.
        String laterArrival = arriveAlice.compareTo(arriveBob) < 0 ? arriveBob : arriveAlice;
        // Determine the earlier leave date of the two.
        String earlierLeave = leaveAlice.compareTo(leaveBob) < 0 ? leaveAlice : leaveBob;

        // Convert the dates to the day of the year.
        int startDay = convertToDateInYear(laterArrival);
        int endDay = convertToDateInYear(earlierLeave);

        // Calculate the total number of days they spend together.
        // Ensure it does not result in a negative number if there is no overlap.
        return Math.max(endDay - startDay + 1, 0);
    }

    /**
     * Converts a date string "MM-DD" to the day of the year.
     *
     * @param date String formatted as "MM-DD".
     * @return The day of the year.
     */
    private int convertToDateInYear(String date) {
        // Parse the month from the date string and adjust for 0-based index.
        int monthIndex = Integer.parseInt(date.substring(0, 2)) - 1;
        int dayOfYear = 0;

        // Add the number of days in the months preceding the given month.
        for (int i = 0; i < monthIndex; ++i) {
            dayOfYear += daysInMonth[i];
        }

        // Add the day of the month to the total.
        dayOfYear += Integer.parseInt(date.substring(3));
        return dayOfYear;
    }
}

```

## C++ Solution

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

class Solution {
public:
    std::vector<int> daysInMonth = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

    // Calculates the number of days Alice and Bob spend together based on their arrival and departure dates
    int countDaysTogether(std::string arriveAlice, std::string leaveAlice, std::string arriveBob, std::string leaveBob) {
        // Find the later of the two arrival dates
        std::string latestArrival = arriveAlice < arriveBob ? arriveBob : arriveAlice;
        // Find the earlier of the two departure dates
        std::string earliestDeparture = leaveAlice < leaveBob ? leaveAlice : leaveBob;
        // Convert dates to day of the year
        int dayOfYearLatestArrival = convertDateToDayOfYear(latestArrival);
        int dayOfYearEarliestDeparture = convertDateToDayOfYear(earliestDeparture);
        // Calculate the number of days they spend together and ensure it's not negative
        return std::max(0, dayOfYearEarliestDeparture - dayOfYearLatestArrival + 1);
    }

    // Converts a date in MM-DD format to a number representing the day of the year
    private:
    int convertDateToDayOfYear(std::string date) {
        int month, day;
        // Parse the month and day from the date string
        sscanf(date.c_str(), "%d-%d", &month, &day);
        int dayOfYear = 0;
        // Accumulate the total number of days from the beginning of the year to the end of the previous month
        for (int i = 0; i < month - 1; ++i) {
            dayOfYear += daysInMonth[i];
        }
        // Add the days in the current month
        dayOfYear += day;
        return dayOfYear;
    }
};

```

## Typescript Solution

```typescript
// An array storing the number of days in each month, assuming it is not a leap year
const daysInMonth = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31];

/**
 * Calculates the number of days Alice and Bob spend together based on their arrival and departure dates
 *
 * @param arriveAlice - The arrival date for Alice in MM-DD format
 * @param leaveAlice - The departure date for Alice in MM-DD format
 * @param arriveBob - The arrival date for Bob in MM-DD format
 * @param leaveBob - The departure date for Bob in MM-DD format
 * @returns The number of days Alice and Bob spend together
 */
function countDaysTogether(arriveAlice: string, leaveAlice: string, arriveBob: string, leaveBob: string): number {
    // Find the later of the two arrival dates
    const latestArrival = arriveAlice < arriveBob ? arriveBob : arriveAlice;
    // Find the earlier of the two departure dates
    const earliestDeparture = leaveAlice < leaveBob ? leaveAlice : leaveBob;
    // Convert dates to day of the year
    const dayOfYearLatestArrival = convertDateToDayOfYear(latestArrival);
    const dayOfYearEarliestDeparture = convertDateToDayOfYear(earliestDeparture);
    // Calculate the number of days they spend together and ensure it's not negative
    return Math.max(0, dayOfYearEarliestDeparture - dayOfYearLatestArrival + 1);
}

/**
 * Converts a date in MM-DD format to a number representing the day of the year
 *
 * @param date - The date in MM-DD format
 * @returns The day of the year based on the date provided
 */
function convertDateToDayOfYear(date: string): number {
    // Parse the month and day from the date string
    const [month, day] = date.split('-').map(Number);
    let dayOfYear = 0;
    // Accumulate the total number of days from the beginning of the year to the end of the previous month
    for (let i = 0; i < month - 1; ++i) {
        dayOfYear += daysInMonth[i];
    }
    // Add the days in the current month
    dayOfYear += day;
    return dayOfYear;
}

// Example usage:
// console.log(countDaysTogether('08-15', '08-18', '08-16', '08-20'));

```

## Time and Space Complexity

The given Python code calculates the number of overlapping days that Alice and Bob spend together based on their arrival and departure dates.

### Time Complexity

The time complexity of this code can be considered to be `O(1)`. Here is the breakdown of operations:

- Two `max` and `min` operations on dates: These operations take constant time since the operations are performed on fixed-length strings (date formats).
- Slicing and accessing elements of the date strings: These operations are also constant in time since dates are fixed-length strings.
- Two `sum` operations: The `days` tuple is a fixed size (12 elements representing the days in each month), so summing over a slice of this tuple is also a constant-time operation.
- Integer conversion and arithmetic operations are again done in constant time.

Since all the operations above are done in constant time and do not depend on the size of the input, the overall time complexity is `O(1)`.

### Space Complexity

The space complexity of the function can also be considered `O(1)` because:

- The tuple `days` has a fixed size of 12.
- There are only a fixed number of integer and string variables, `a`, `b`, `x`, and `y`.
- No additional data structures that grow with the input size are being used.

Therefore, the amount of memory used by the program is constant, irrespective of the input, leading to a space complexity of `O(1)`.