Prefix & Suffix Sum Assessment

Instructions:

Tool used: VSCode or Intellij Idea with proper environment setup

Test date: 4th Jan, 2025

Test mode: Offline (A-006)

Reporting time: 9:45 AM - 10:00 AM

Test time: 10:00 AM - 11:30 AM

Submission time: 11:30 AM - 12:00 PM

Solution discussion time: 1:15 PM - 2:45 PM

Offline Assessment Guidelines:

- **Internet Usage:**
  - Ensure that the internet is closed during the assessment. The assessment will be conducted offline using an IDE.
  - Use of any online resources is prohibited.
- **Mobile Phones:**
  - Switch off your mobile phone.
  - Place your mobile phone inside your bag, and keep your bag outside the assessment venue.
- **Items to Carry:**
  - Bring your **laptop** and **charger**.
  - Carry a **pen**, **water bottle**, and **A4 sheets** for dry run submission.

Marking scheme:

- **Clean & Concise Code:** Proper naming conventions, modular code, reusable functions.
- **Code correctness:** Does the solution solve the problem as expected?
- **Efficiency:** time and space complexity, optimal solution.
- **Edge Cases:** handling of boundary conditions and exceptional inputs.

- **Time and Space Complexity:** Document the complexity analysis of your solution in the code using comments.
- **Dry run:** Please perform a dry run of the program and submit the results as a hard copy. Ensure that your dry run includes step-by-step tracing of the input, intermediate computations, and the final output.

Steps for GitHub Repo Usage and Submission:

- **Fork the repository:**
    - Fork the repo from: https://github.com/codetatvaacademy/GU-Technical-Training-2026
- **Create Your Branch:**
    - Do not directly use the `main` branch.
    - Create a new branch in your forked repository for your work.
- **Create a Directory for Your Files:**
    - Inside your branch, create a directory structure like:
        - `Aniruddha-M-Agrawal/PrefixAndSuffixSumAssesment/Problem1.java`
        - `Aniruddha-M-Agrawal/PrefixAndSuffixSumAssesment/Problem2.java`
    - Each file should correspond to an individual problem.
- **Submit a Pull Request:**
    - Once you have completed the required work, submit a pull request from your branch to the `main` branch of the original repository.
- **Review and Merge:**
    - I will review your code and merge it into the `main` branch after reviewing the solution.
    - Marks will be assigned based on the marking scheme.

Problem 1:

In a small village, there's a legendary seesaw located at the center of the town. The villagers believe that if someone can balance the seesaw perfectly with their weights, they will receive immense wisdom and clarity in decision-making.

One day, a curious traveler named Alex arrives in the village. He is given an array of weights, representing villagers lined up on the seesaw. Alex's task is to find the **middleIndex**, where the seesaw is perfectly balanced. The rules for balancing the seesaw are:

1. The sum of weights on the **left side** of the middleIndex must equal the sum of weights on the **right side**.
2. If Alex considers the first position (index 0), the left side has no weights, so the sum is **0**.
3. Similarly, if Alex considers the last position (index n−1n - 1n−1), the right side sum is **0**.

Alex wants to find the **leftmost middleIndex** where the seesaw balances. If no such index exists, he will consider the task impossible and return **-1**.

Input:

- Alex receives an integer array, **nums**, representing the weights of villagers.

Output:

- If Alex finds the seesaw's balance point, he returns the **leftmost middleIndex**.
- If no such index exists, he returns **-1**.

Example 1:

Input:
`nums = [2, 3, -1, 8, 4]`

Output:
`3`

Explanation:

- Left of index `3`: 2+3+(−1)=4

- Right of index 3: 4
- The seesaw balances perfectly at index 3.

Example 2:

Input:
`nums = [1, -1, 4]`

Output:
`2`

Explanation:

- Left of index 2: $1+(-1)=0$
- Right of index 2: 0
- The seesaw balances perfectly at index 2.

Example 3:

Input:
`nums = [2, 5]`

Output:
`-1`

Explanation:

- No index satisfies the balance condition.

Constraints:

1. $1 \le nums.length \le 1000$
2. $-1000 \le nums[i] \le 1000$

Problem 2:

In the bustling town of Flytown, a new airline company has introduced a unique booking system to manage its flight reservations. The town's people, excited about the new service, start reserving seats for different flights. However, the airline faces a small challenge—they need to calculate the total number of seats reserved for each flight efficiently.

Here’s how the system works:

1. There are **n flights** labeled from **1** to **n**.
2. Customers make reservations for flights in **ranges**, where:
    - Each booking specifies a starting flight (**firsti**) and an ending flight (**lasti**).
    - For all flights between **firsti** and **lasti** (inclusive), the same number of seats (**seatsi**) is reserved.

The airline now needs to calculate the **total seats reserved for each flight**.

Input:

The airline provides:

1. **n**: The total number of flights.
2. **bookings**: A list of bookings, where each booking is represented as [firsti,lasti,seatsi]

Output:

The airline requires:

- An array **answer** of length **n**, where **answer[i]** represents the total number of seats reserved for flight iii (1-based index).

Example 1:

Input:
n=5
bookings=[[1,2,10],[2,3,20],[2,5,25]]

Output:
[10,55,45,25,25]

Explanation:

- For flight 1: 10 seats are reserved.
- For flight 2: 10+20+25=55 seats are reserved.
- For flight 3: 20+25=45 seats are reserved.
- For flight 4: 25 seats are reserved.
- For flight 5: 25 seats are reserved.

Example 2:

Input:
n=3
bookings=[[1,3,5],[2,2,10]]

Output:
[5,15,55]

Explanation:

- For flight 1: 5 seats are reserved.
- For flight 2: 5+10=15 seats are reserved.
- For flight 3: 55 seats are reserved.

Constraints:

1 <= n <= 2 * 10^4

1 <= bookings.length <= 2 * 10^4

bookings[i].length == 3

1 <= firsti <= lasti <= n

1 <= seatsi <= 10^4

Problem 3:

Given a 2D matrix matrix, handle multiple queries of the following type:

Calculate the sum of the elements of matrix inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).

Implement the NumMatrix class:

NumMatrix(int[][] matrix) Initializes the object with the integer matrix matrix.

int sumRegion(int row1, int col1, int row2, int col2) Returns the sum of the elements of matrix inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).

You must design an algorithm where sumRegion works on O(1) time complexity.

Example 1:

Input

["NumMatrix", "sumRegion", "sumRegion", "sumRegion"]

[[[[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4, 1, 0, 1, 7], [1, 0, 3, 0, 5]]], [2, 1, 4, 3], [1, 1, 2, 2], [1, 2, 2, 4]]

Output

[null, 8, 11, 12]

Explanation

NumMatrix numMatrix = new NumMatrix([[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4, 1, 0, 1, 7], [1, 0, 3, 0, 5]]);

numMatrix.sumRegion(2, 1, 4, 3); // return 8 (i.e sum of the red rectangle)

numMatrix.sumRegion(1, 1, 2, 2); // return 11 (i.e sum of the green rectangle)

numMatrix.sumRegion(1, 2, 2, 4); // return 12 (i.e sum of the blue rectangle)

Constraints:

m == matrix.length

n == matrix[i].length

1 <= m, n <= 200

-10^4 <= matrix[i][j] <= 10^4

0 <= row1 <= row2 < m

0 <= col1 <= col2 < n

At most 10^4 calls will be made to sumRegion.

```
class NumMatrix {

   public NumMatrix(int[][] matrix) {

   }



   public int sumRegion(int row1, int col1, int row2, int col2) {

   }

}

/**

* Your NumMatrix object will be instantiated and called as such:

* NumMatrix obj = new NumMatrix(matrix);

* int param_1 = obj.sumRegion(row1,col1,row2,col2);

*/
```

Please implement the complete code in your IDE, ensuring it includes your best input-output format. Use the provided code structure for this problem and include the main function to execute the program directly from your IDE.