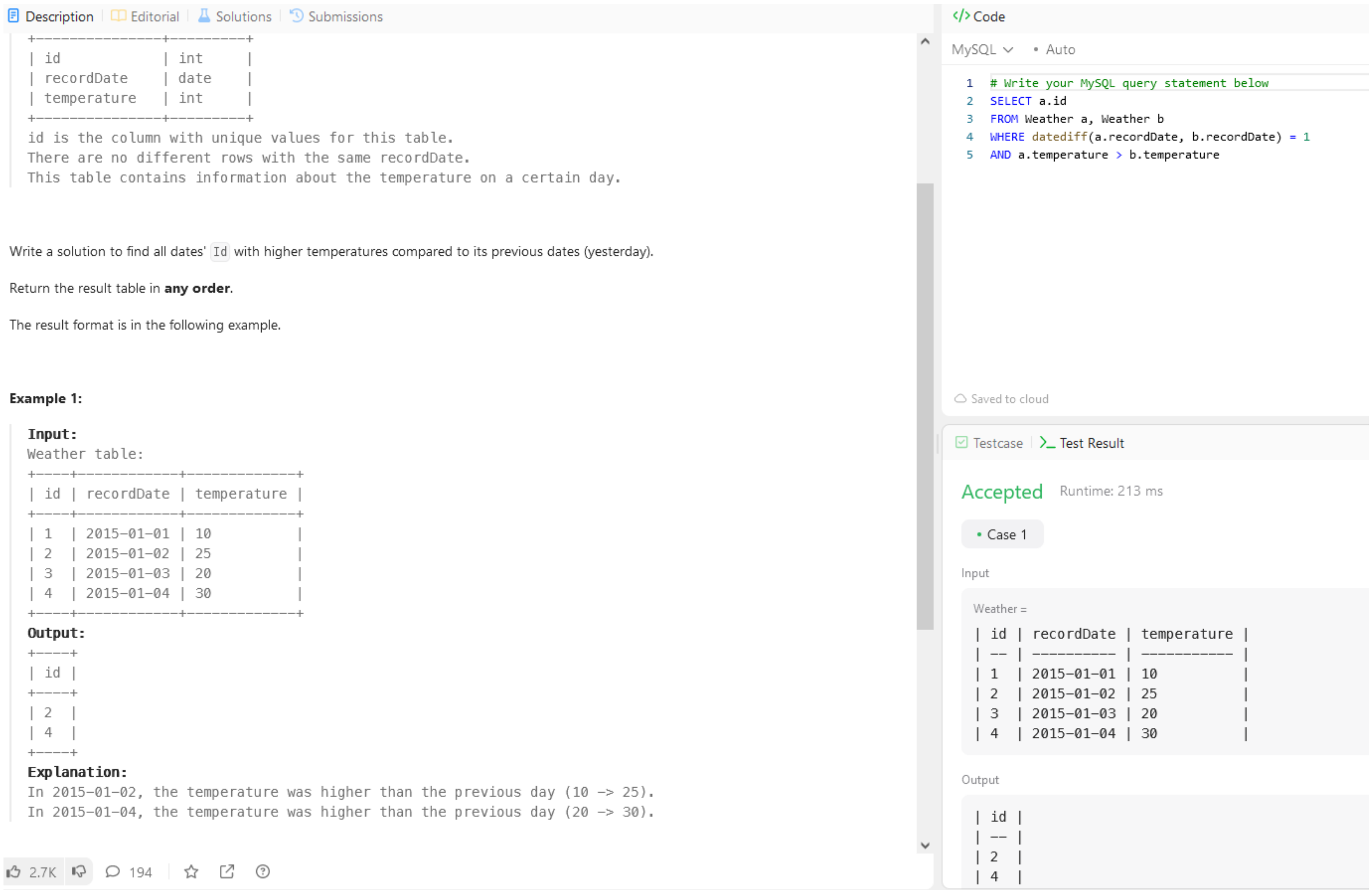


```
+---------------+---------+
| id            | int     |
| recordDate    | date    |
| temperature   | int     |
+---------------+---------+
id is the column with unique values for this table.
There are no different rows with the same recordDate.
This table contains information about the temperature on a certain day.
```

Write a solution to find all dates' `Id` with higher temperatures compared to its previous dates (yesterday).

Return the result table in **any order**.

The result format is in the following example.

Example 1:

```
Input:
Weather table:
+----+------------+-------------+
| id | recordDate | temperature |
+----+------------+-------------+
| 1  | 2015-01-01 | 10          |
| 2  | 2015-01-02 | 25          |
| 3  | 2015-01-03 | 20          |
| 4  | 2015-01-04 | 30          |
+----+------------+-------------+
Output:
+----+
| id |
+----+
| 2  |
| 4  |
+----+
Explanation:
In 2015-01-02, the temperature was higher than the previous day (10 -> 25).
In 2015-01-04, the temperature was higher than the previous day (20 -> 30).
```

```sql
# Write your MySQL query statement below
SELECT a.id
FROM Weather a, Weather b
WHERE datediff(a.recordDate, b.recordDate) = 1
AND a.temperature > b.temperature
```

Accepted Runtime: 213 ms

Case 1

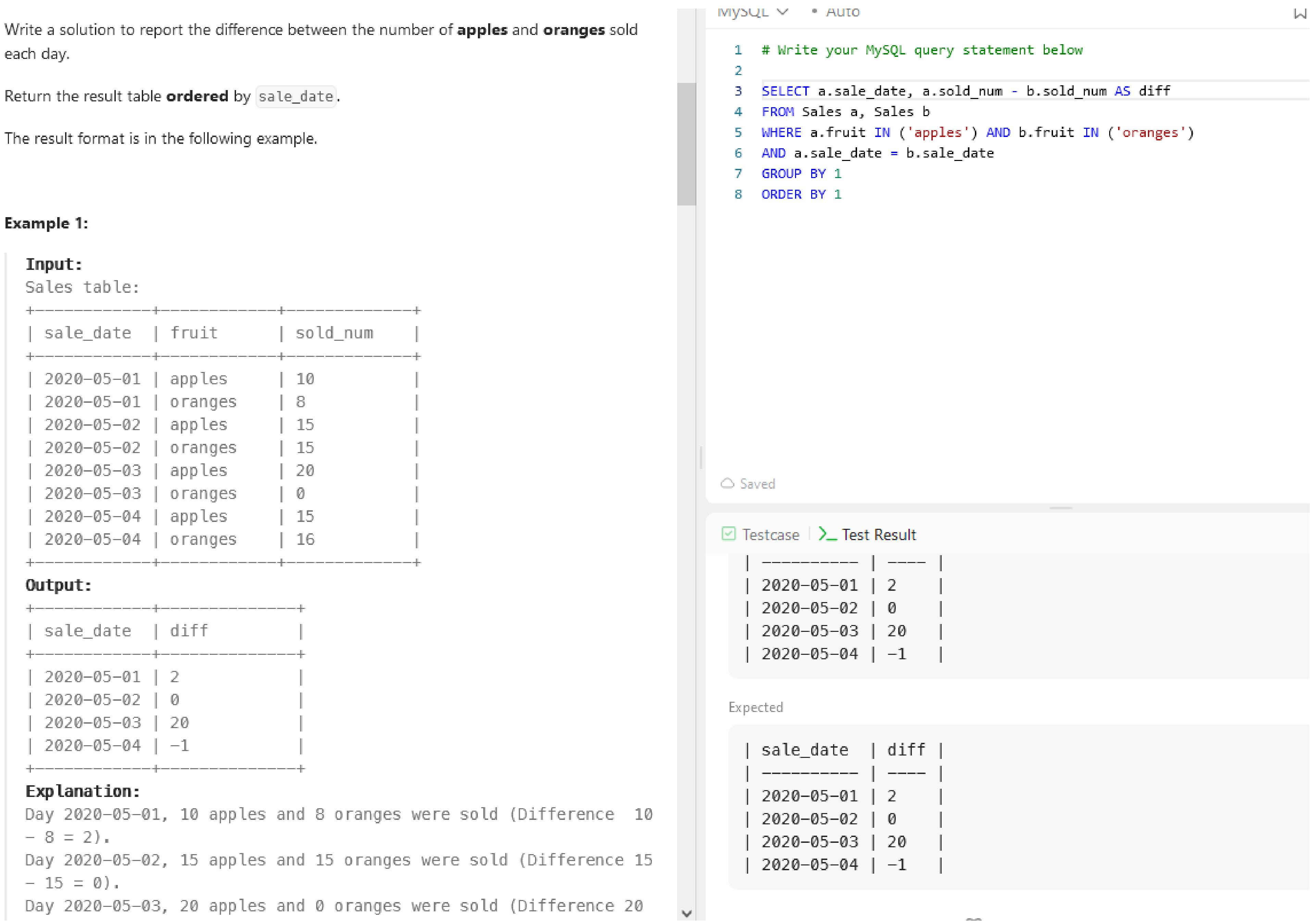
Input

Weather =

```
| id | recordDate | temperature |
| -- | ---------- | ----------- |
| 1  | 2015-01-01 | 10          |
| 2  | 2015-01-02 | 25          |
| 3  | 2015-01-03 | 20          |
| 4  | 2015-01-04 | 30          |
```

Output

```
| id |
| -- |
| 2  |
| 4  |
```



Write a solution to report the difference between the number of **apples** and **oranges** sold each day.

Return the result table **ordered** by `sale_date`.

The result format is in the following example.

Example 1:

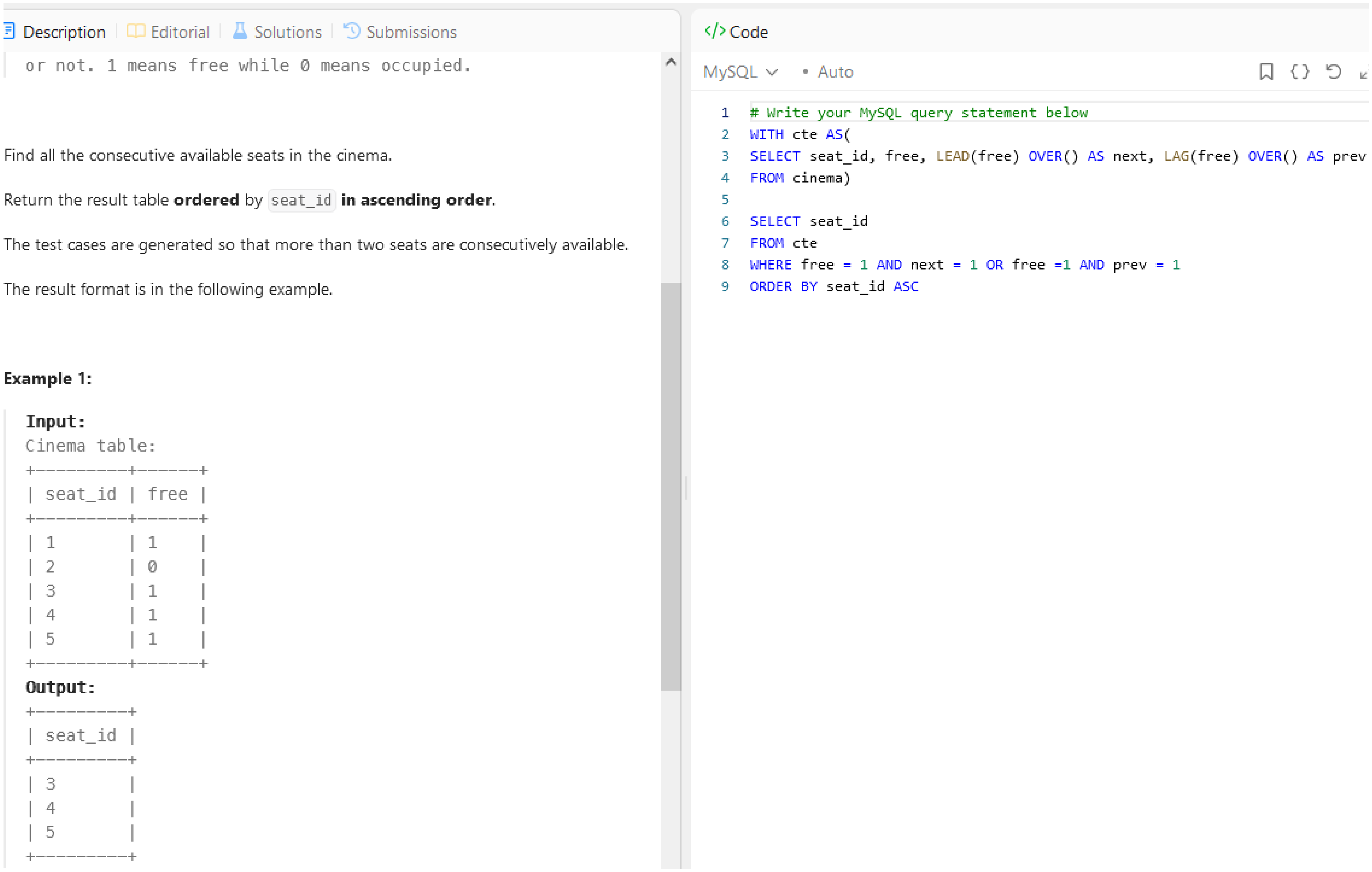
```
Input:
Sales table:
+------------+------------+-------------+
| sale_date  | fruit      | sold_num    |
+------------+------------+-------------+
| 2020-05-01 | apples     | 10          |
| 2020-05-01 | oranges    | 8           |
| 2020-05-02 | apples     | 15          |
| 2020-05-02 | oranges    | 15          |
| 2020-05-03 | apples     | 20          |
| 2020-05-03 | oranges    | 0           |
| 2020-05-04 | apples     | 15          |
| 2020-05-04 | oranges    | 16          |
+------------+------------+-------------+
Output:
+------------+--------------+
| sale_date  | diff         |
+------------+--------------+
| 2020-05-01 | 2            |
| 2020-05-02 | 0            |
| 2020-05-03 | 20           |
| 2020-05-04 | -1           |
+------------+--------------+
Explanation:
Day 2020-05-01, 10 apples and 8 oranges were sold (Difference  10
- 8 = 2).
Day 2020-05-02, 15 apples and 15 oranges were sold (Difference 15
- 15 = 0).
Day 2020-05-03, 20 apples and 0 oranges were sold (Difference 20
```

```sql
# Write your MySQL query statement below

SELECT a.sale_date, a.sold_num - b.sold_num AS diff
FROM Sales a, Sales b
WHERE a.fruit IN ('apples') AND b.fruit IN ('oranges')
AND a.sale_date = b.sale_date
GROUP BY 1
ORDER BY 1
```

```
| ---------- | ---- |
| 2020-05-01 | 2    |
| 2020-05-02 | 0    |
| 2020-05-03 | 20   |
| 2020-05-04 | -1   |
```

Expected

```
| sale_date  | diff |
| ---------- | ---- |
| 2020-05-01 | 2    |
| 2020-05-02 | 0    |
| 2020-05-03 | 20   |
| 2020-05-04 | -1   |
```


Description | Editorial | Solutions | Submissions

```
or not. 1 means free while 0 means occupied.
```

Find all the consecutive available seats in the cinema.

Return the result table **ordered** by `seat_id` **in ascending order**.

The test cases are generated so that more than two seats are consecutively available.

The result format is in the following example.

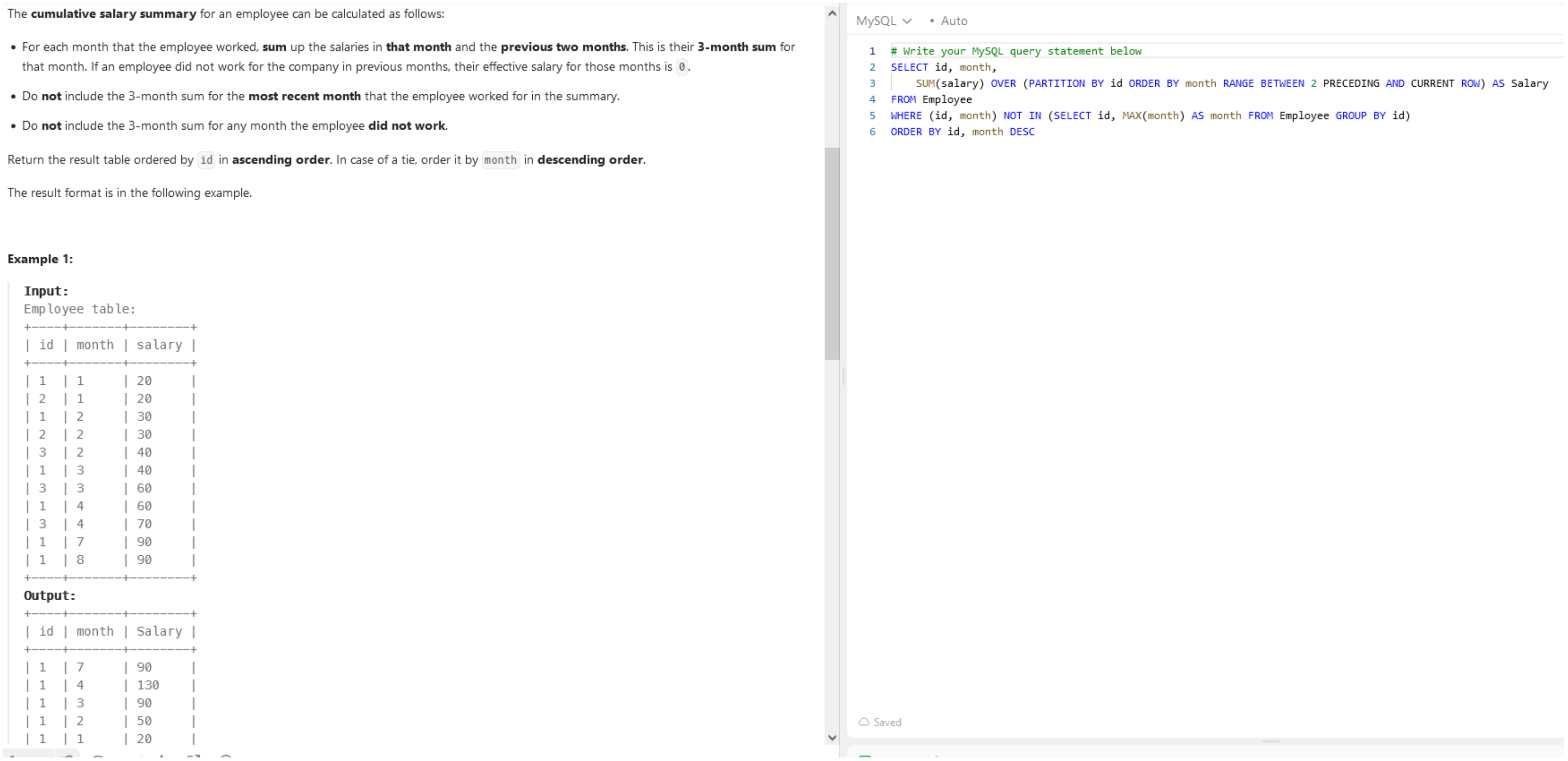
Example 1:

```
Input:
Cinema table:
+---------+------+
| seat_id | free |
+---------+------+
| 1       | 1    |
| 2       | 0    |
| 3       | 1    |
| 4       | 1    |
| 5       | 1    |
+---------+------+
Output:
+---------+
| seat_id |
+---------+
| 3       |
| 4       |
| 5       |
+---------+
```

Code

MySQL • Auto

```sql
# Write your MySQL query statement below
WITH cte AS(
SELECT seat_id, free, LEAD(free) OVER() AS next, LAG(free) OVER() AS prev
FROM cinema)

SELECT seat_id
FROM cte
WHERE free = 1 AND next = 1 OR free =1 AND prev = 1
ORDER BY seat_id ASC
```



The **cumulative salary summary** for an employee can be calculated as follows:

- For each month that the employee worked, **sum** up the salaries in **that month** and the **previous two months**. This is their **3-month sum** for that month. If an employee did not work for the company in previous months, their effective salary for those months is `0`.
- Do **not** include the 3-month sum for the **most recent month** that the employee worked for in the summary.
- Do **not** include the 3-month sum for any month the employee **did not work**.

Return the result table ordered by `id` in **ascending order**. In case of a tie, order it by `month` in **descending order**.

The result format is in the following example.

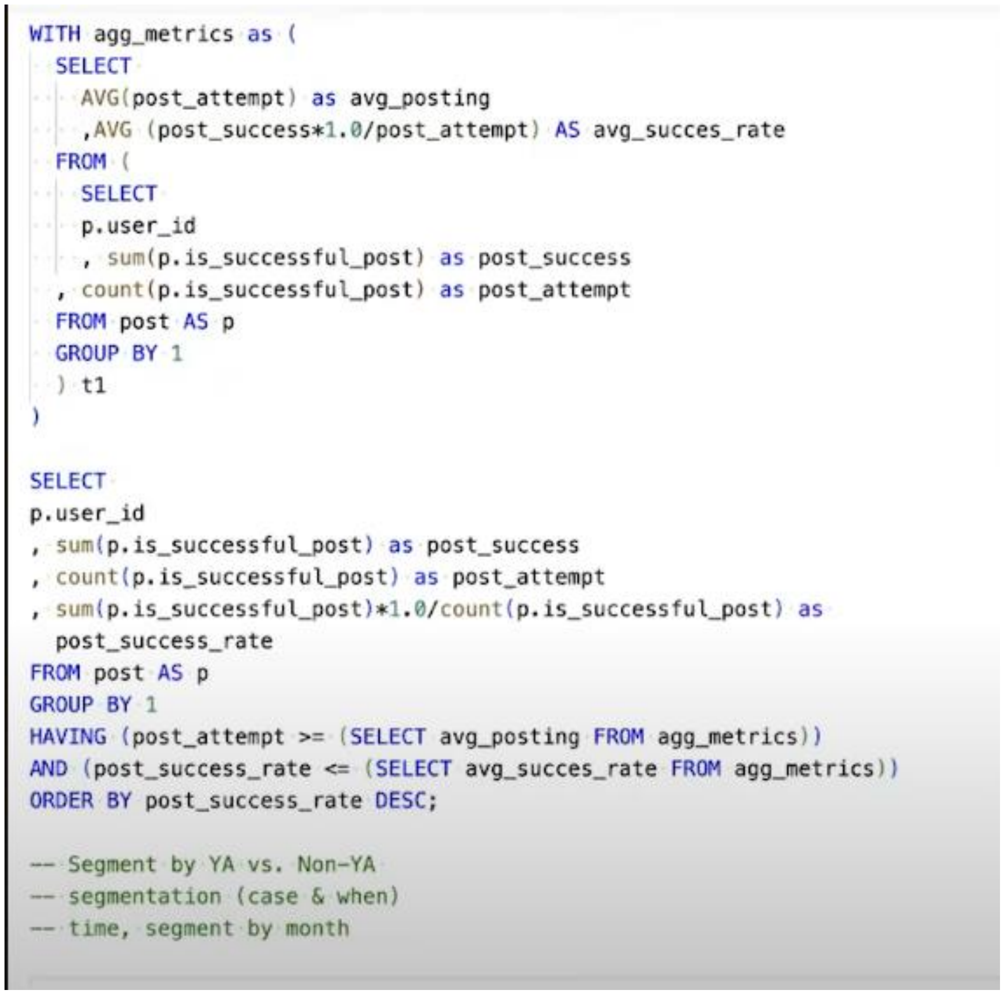
Example 1:

```
Input:
Employee table:
+----+-------+--------+
| id | month | salary |
+----+-------+--------+
| 1  | 1     | 20     |
| 2  | 1     | 20     |
| 1  | 2     | 30     |
| 2  | 2     | 30     |
| 3  | 2     | 40     |
| 1  | 3     | 40     |
| 3  | 3     | 60     |
| 1  | 4     | 60     |
| 3  | 4     | 70     |
| 1  | 7     | 90     |
| 1  | 8     | 90     |
+----+-------+--------+
Output:
+----+-------+--------+
| id | month | Salary |
+----+-------+--------+
| 1  | 7     | 90     |
| 1  | 4     | 130    |
| 1  | 3     | 90     |
| 1  | 2     | 50     |
| 1  | 1     | 20     |
```

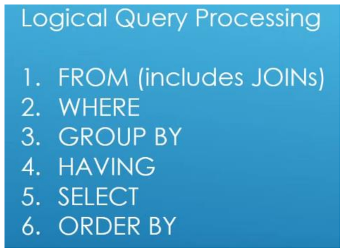
MySQL • Auto

```sql
# Write your MySQL query statement below
SELECT id, month,
    SUM(salary) OVER (PARTITION BY id ORDER BY month RANGE BETWEEN 2 PRECEDING AND CURRENT ROW) AS Salary
FROM Employee
WHERE (id, month) NOT IN (SELECT id, MAX(month) AS month FROM Employee GROUP BY id)
ORDER BY id, month DESC
```

Saved

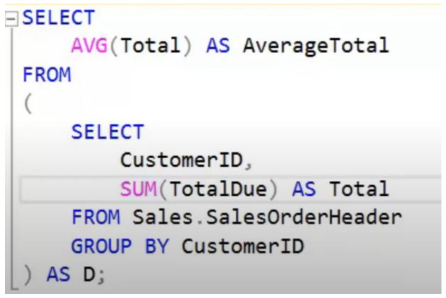


```
WITH agg_metrics as (
  SELECT
    AVG(post_attempt) as avg_posting
    ,AVG (post_success*1.0/post_attempt) AS avg_succes_rate
  FROM (
    SELECT
    p.user_id
    , sum(p.is_successful_post) as post_success
  , count(p.is_successful_post) as post_attempt
  FROM post AS p
  GROUP BY 1
  ) t1
)

SELECT
p.user_id
, sum(p.is_successful_post) as post_success
, count(p.is_successful_post) as post_attempt
, sum(p.is_successful_post)*1.0/count(p.is_successful_post) as
  post_success_rate
FROM post AS p
GROUP BY 1
HAVING (post_attempt >= (SELECT avg_posting FROM agg_metrics))
AND (post_success_rate <= (SELECT avg_succes_rate FROM agg_metrics))
ORDER BY post_success_rate DESC;

-- Segment by YA vs. Non-YA
-- segmentation (case & when)
-- time, segment by month
```

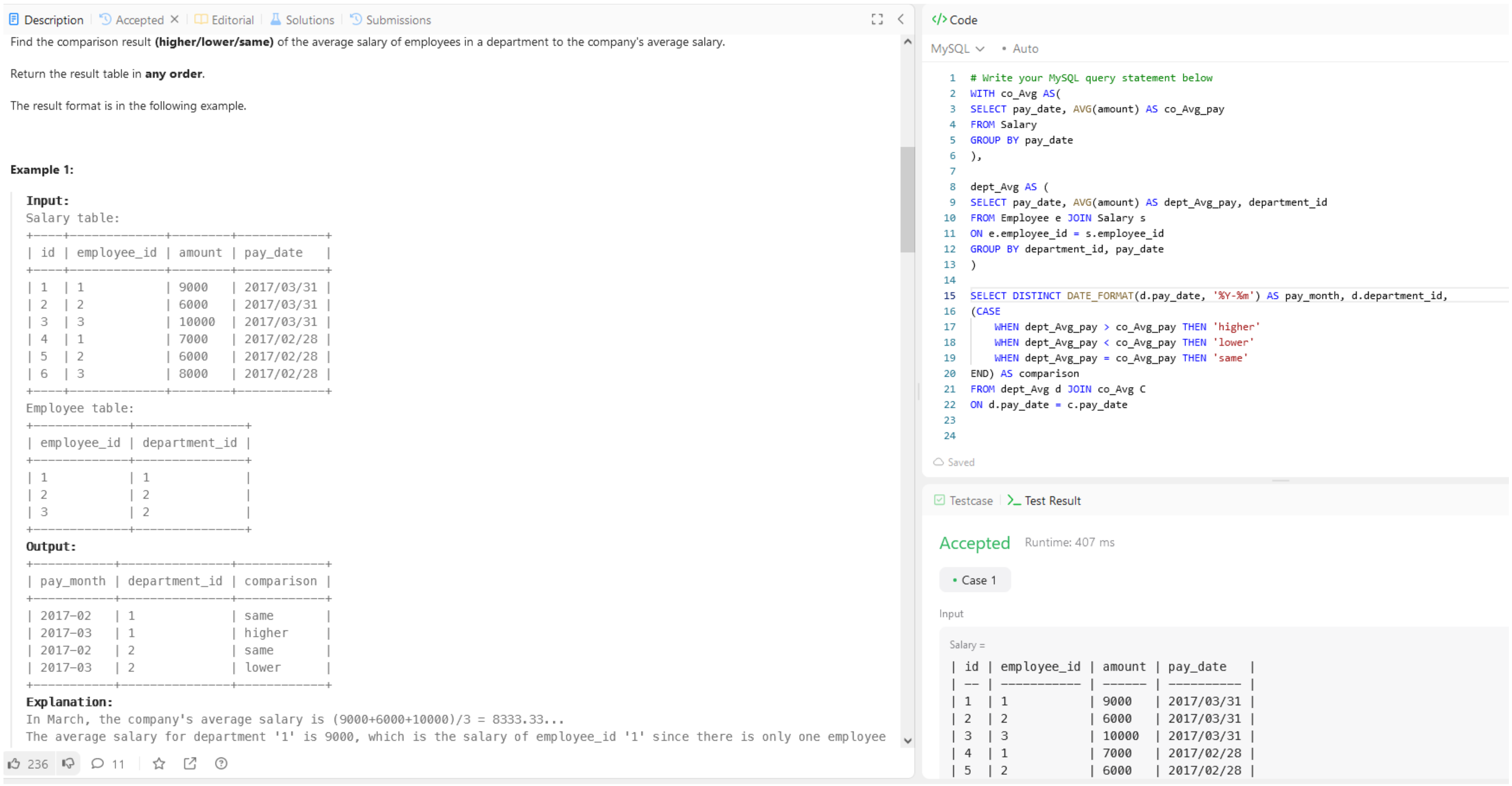


Logical Query Processing

1. FROM (includes JOINs)
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY



```
SELECT
    AVG(Total) AS AverageTotal
FROM
(
    SELECT
        CustomerID,
        SUM(TotalDue) AS Total
    FROM Sales.SalesOrderHeader
    GROUP BY CustomerID
) AS D;
```



Find the comparison result **(higher/lower/same)** of the average salary of employees in a department to the company's average salary.

Return the result table in **any order**.

The result format is in the following example.

Example 1:

```
Input:
Salary table:
+----+-------------+--------+------------+
| id | employee_id | amount | pay_date   |
+----+-------------+--------+------------+
| 1  | 1           | 9000   | 2017/03/31 |
| 2  | 2           | 6000   | 2017/03/31 |
| 3  | 3           | 10000  | 2017/03/31 |
| 4  | 1           | 7000   | 2017/02/28 |
| 5  | 2           | 6000   | 2017/02/28 |
| 6  | 3           | 8000   | 2017/02/28 |
+----+-------------+--------+------------+
Employee table:
+-------------+---------------+
| employee_id | department_id |
+-------------+---------------+
| 1           | 1             |
| 2           | 2             |
| 3           | 2             |
+-------------+---------------+
Output:
+-----------+---------------+------------+
| pay_month | department_id | comparison |
+-----------+---------------+------------+
| 2017-02   | 1             | same       |
| 2017-03   | 1             | higher     |
| 2017-02   | 2             | same       |
| 2017-03   | 2             | lower      |
+-----------+---------------+------------+
Explanation:
In March, the company's average salary is (9000+6000+10000)/3 = 8333.33...
The average salary for department '1' is 9000, which is the salary of employee_id '1' since there is only one employee
```

```sql
# Write your MySQL query statement below
WITH co_Avg AS(
SELECT pay_date, AVG(amount) AS co_Avg_pay
FROM Salary
GROUP BY pay_date
),

dept_Avg AS (
SELECT pay_date, AVG(amount) AS dept_Avg_pay, department_id
FROM Employee e JOIN Salary s
ON e.employee_id = s.employee_id
GROUP BY department_id, pay_date
)

SELECT DISTINCT DATE_FORMAT(d.pay_date, '%Y-%m') AS pay_month, d.department_id,
(CASE
    WHEN dept_Avg_pay > co_Avg_pay THEN 'higher'
    WHEN dept_Avg_pay < co_Avg_pay THEN 'lower'
    WHEN dept_Avg_pay = co_Avg_pay THEN 'same'
END) AS comparison
FROM dept_Avg d JOIN co_Avg C
ON d.pay_date = c.pay_date
```

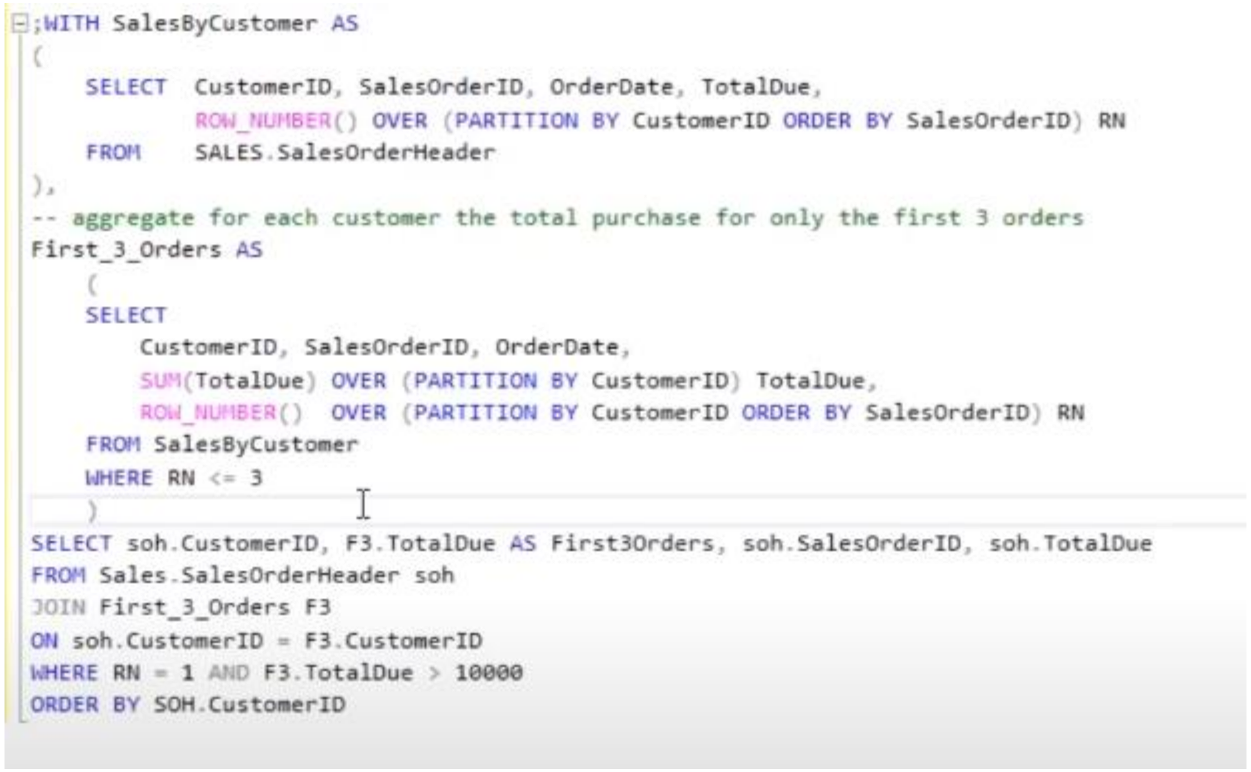
Accepted Runtime: 407 ms

Case 1

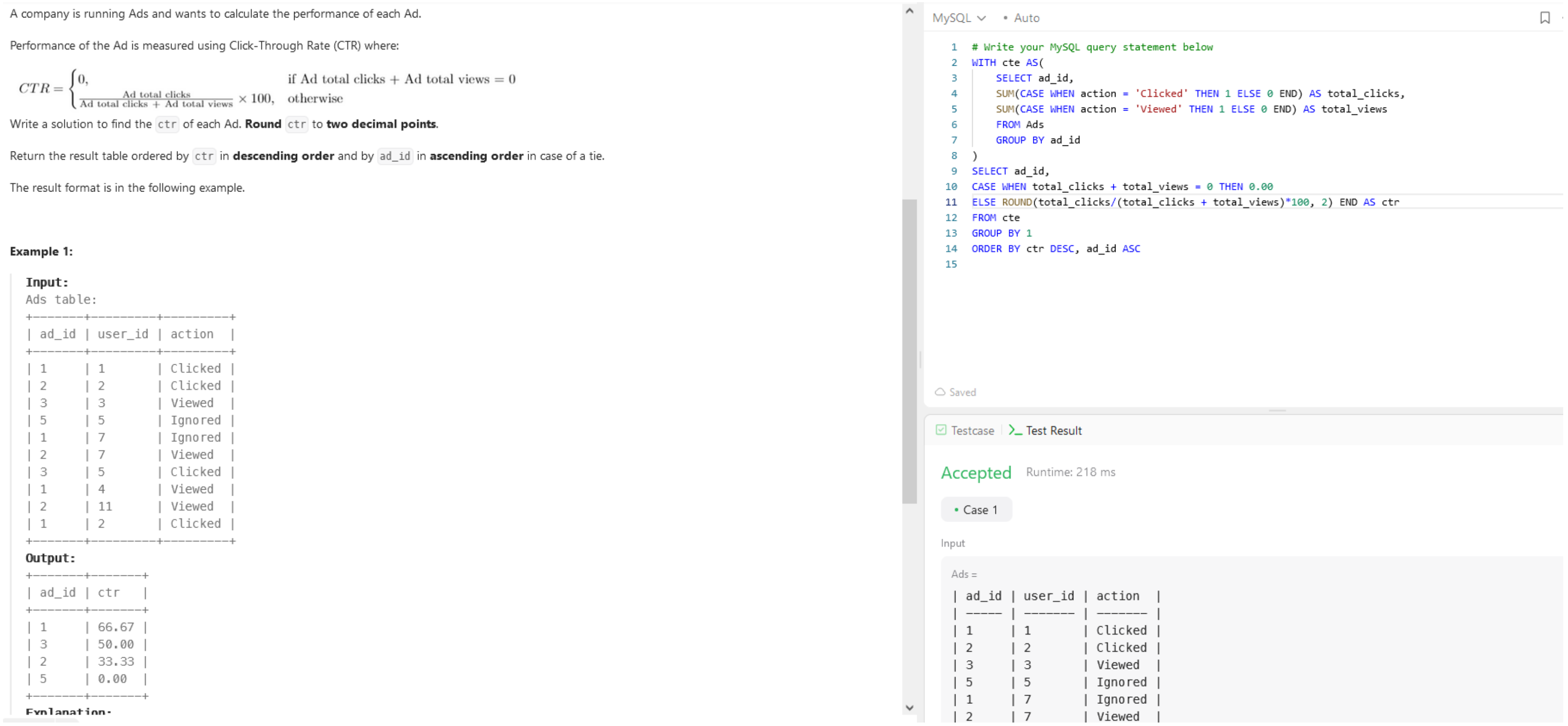
Input

```
Salary =
| id | employee_id | amount | pay_date   |
| -- | ----------- | ------ | ---------- |
| 1  | 1           | 9000   | 2017/03/31 |
| 2  | 2           | 6000   | 2017/03/31 |
| 3  | 3           | 10000  | 2017/03/31 |
| 4  | 1           | 7000   | 2017/02/28 |
| 5  | 2           | 6000   | 2017/02/28 |
```

```sql
SELECT b.book_id, name, COALESCE(SUM(quantity),0)
FROM Orders o RIGHT JOIN Books b
ON o.book_id = b.book_id
GROUP BY o.book_id
#HAVING COALESCE(SUM(quantity),0) < 10
```



```
;WITH SalesByCustomer AS
(
    SELECT  CustomerID, SalesOrderID, OrderDate, TotalDue,
            ROW_NUMBER() OVER (PARTITION BY CustomerID ORDER BY SalesOrderID) RN
    FROM    SALES.SalesOrderHeader
),
-- aggregate for each customer the total purchase for only the first 3 orders
First_3_Orders AS
    (
    SELECT
        CustomerID, SalesOrderID, OrderDate,
        SUM(TotalDue) OVER (PARTITION BY CustomerID) TotalDue,
        ROW_NUMBER()  OVER (PARTITION BY CustomerID ORDER BY SalesOrderID) RN
    FROM SalesByCustomer
    WHERE RN <= 3
    )
SELECT soh.CustomerID, F3.TotalDue AS First3Orders, soh.SalesOrderID, soh.TotalDue
FROM Sales.SalesOrderHeader soh
JOIN First_3_Orders F3
ON soh.CustomerID = F3.CustomerID
WHERE RN = 1 AND F3.TotalDue > 10000
ORDER BY SOH.CustomerID
```



A company is running Ads and wants to calculate the performance of each Ad.

Performance of the Ad is measured using Click-Through Rate (CTR) where:

$$CTR = \begin{cases} 0, & \text{if Ad total clicks} + \text{Ad total views} = 0 \\ \frac{\text{Ad total clicks}}{\text{Ad total clicks} + \text{Ad total views}} \times 100, & \text{otherwise} \end{cases}$$

Write a solution to find the `ctr` of each Ad. **Round** `ctr` to **two decimal points**.

Return the result table ordered by `ctr` in **descending order** and by `ad_id` in **ascending order** in case of a tie.

The result format is in the following example.

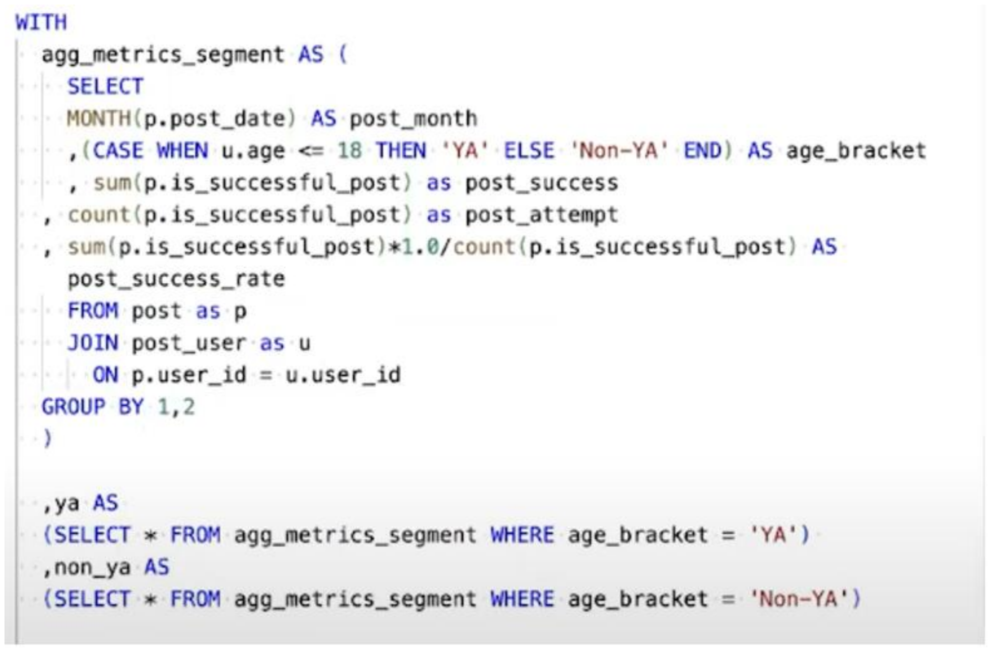
Example 1:

```
Input:
Ads table:
+-------+---------+---------+
| ad_id | user_id | action  |
+-------+---------+---------+
| 1     | 1       | Clicked |
| 2     | 2       | Clicked |
| 3     | 3       | Viewed  |
| 5     | 5       | Ignored |
| 1     | 7       | Ignored |
| 2     | 7       | Viewed  |
| 3     | 5       | Clicked |
| 1     | 4       | Viewed  |
| 2     | 11      | Viewed  |
| 1     | 2       | Clicked |
+-------+---------+---------+
Output:
+-------+-------+
| ad_id | ctr   |
+-------+-------+
| 1     | 66.67 |
| 3     | 50.00 |
| 2     | 33.33 |
| 5     | 0.00  |
+-------+-------+
```

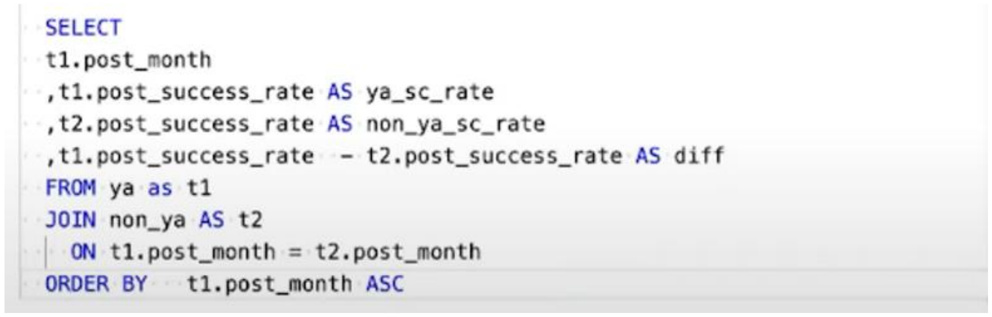
```sql
# Write your MySQL query statement below
WITH cte AS(
    SELECT ad_id,
    SUM(CASE WHEN action = 'Clicked' THEN 1 ELSE 0 END) AS total_clicks,
    SUM(CASE WHEN action = 'Viewed' THEN 1 ELSE 0 END) AS total_views
    FROM Ads
    GROUP BY ad_id
)
SELECT ad_id,
CASE WHEN total_clicks + total_views = 0 THEN 0.00
ELSE ROUND(total_clicks/(total_clicks + total_views)*100, 2) END AS ctr
FROM cte
GROUP BY 1
ORDER BY ctr DESC, ad_id ASC
```

Accepted Runtime: 218 ms

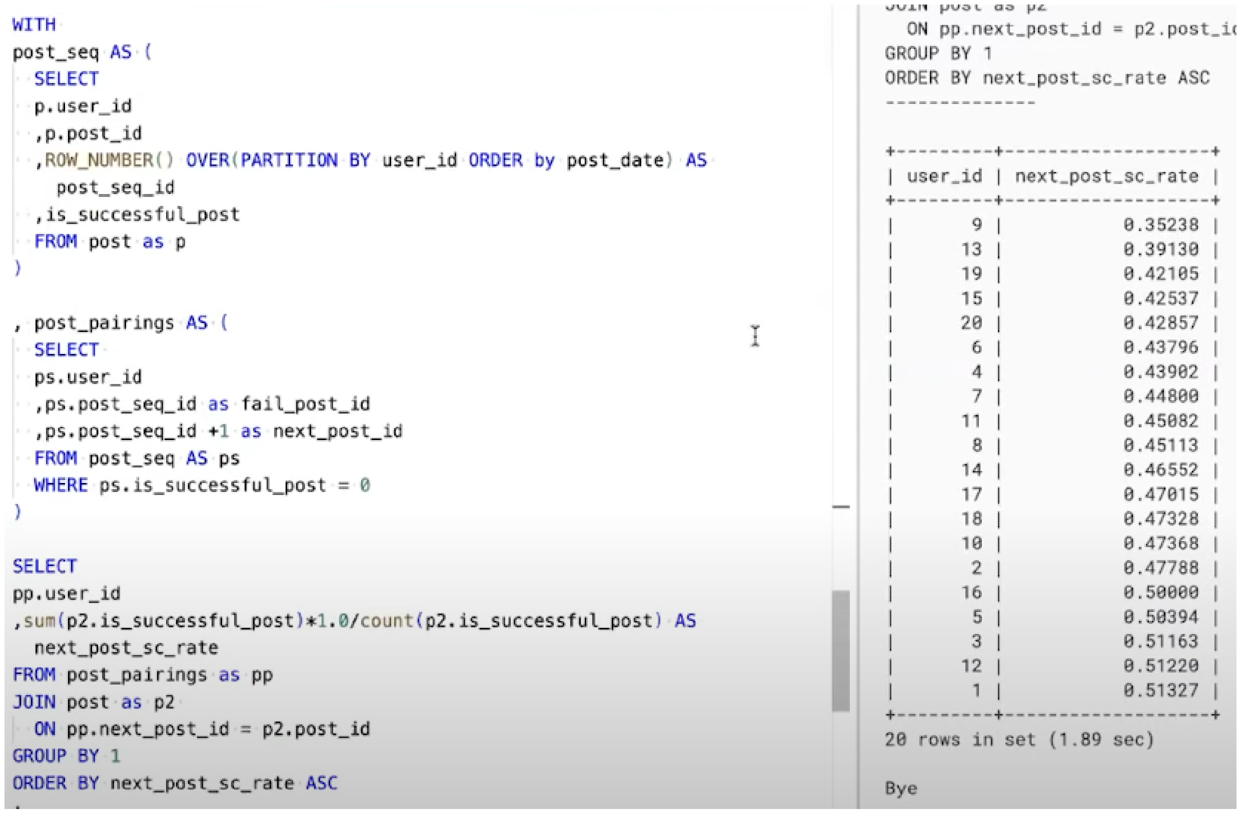
```
Ads =
| ad_id | user_id | action  |
| ----- | ------- | ------- |
| 1     | 1       | Clicked |
| 2     | 2       | Clicked |
| 3     | 3       | Viewed  |
| 5     | 5       | Ignored |
| 1     | 7       | Ignored |
| 2     | 7       | Viewed  |
```



```sql
WITH
 agg_metrics_segment AS (
   SELECT
   MONTH(p.post_date) AS post_month
   ,(CASE WHEN u.age <= 18 THEN 'YA' ELSE 'Non-YA' END) AS age_bracket
   , sum(p.is_successful_post) as post_success
 , count(p.is_successful_post) as post_attempt
 , sum(p.is_successful_post)*1.0/count(p.is_successful_post) AS
   post_success_rate
   FROM post as p
   JOIN post_user as u
     ON p.user_id = u.user_id
 GROUP BY 1,2
 )

 ,ya AS
 (SELECT * FROM agg_metrics_segment WHERE age_bracket = 'YA')
 ,non_ya AS
 (SELECT * FROM agg_metrics_segment WHERE age_bracket = 'Non-YA')
```

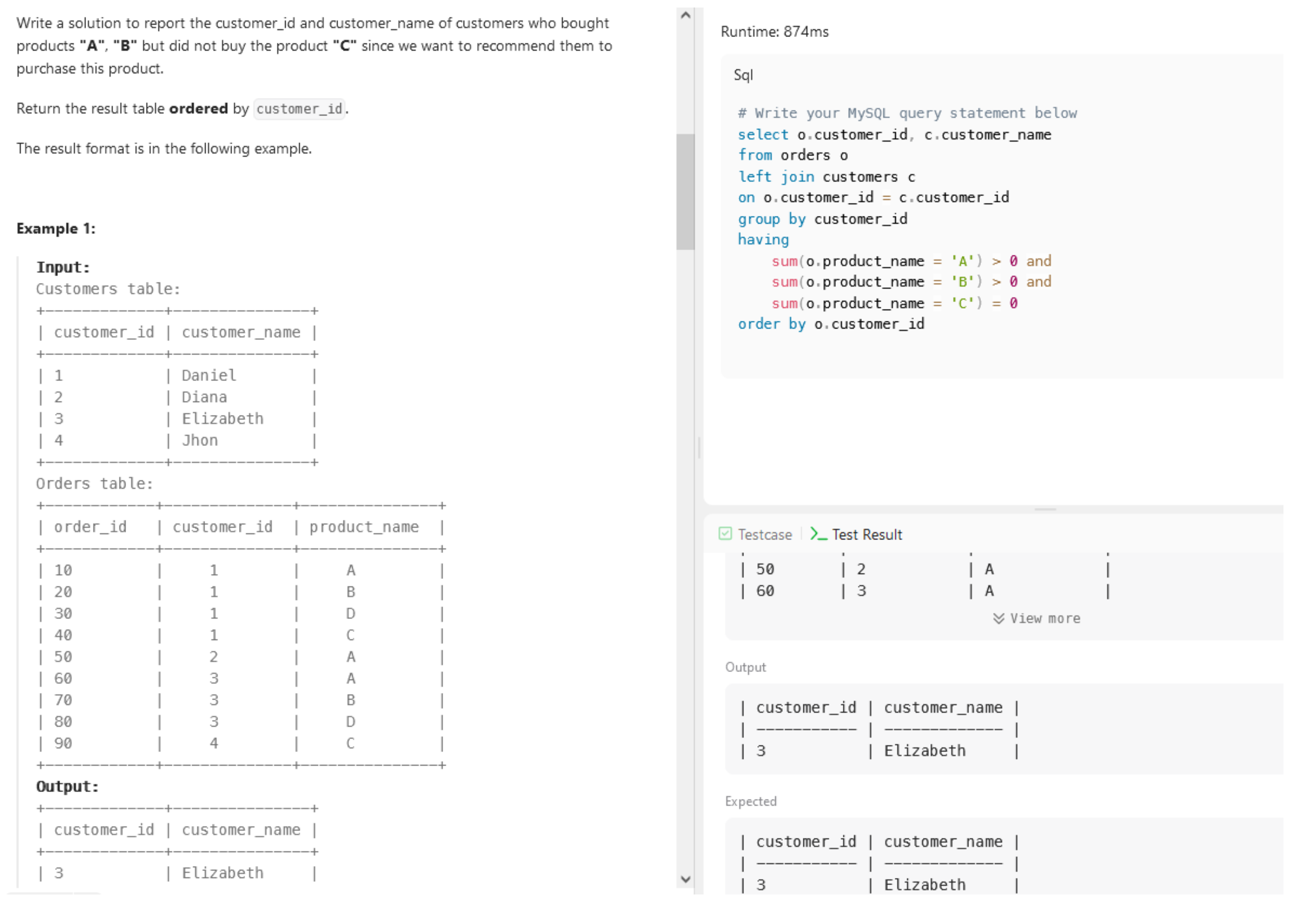


```sql
 SELECT
 t1.post_month
 ,t1.post_success_rate AS ya_sc_rate
 ,t2.post_success_rate AS non_ya_sc_rate
 ,t1.post_success_rate  - t2.post_success_rate AS diff
 FROM ya as t1
 JOIN non_ya AS t2
   ON t1.post_month = t2.post_month
 ORDER BY   t1.post_month ASC
```



```sql
WITH
post_seq AS (
  SELECT
  p.user_id
  ,p.post_id
  ,ROW_NUMBER() OVER(PARTITION BY user_id ORDER by post_date) AS
    post_seq_id
  ,is_successful_post
  FROM post as p
)

, post_pairings AS (
  SELECT
  ps.user_id
  ,ps.post_seq_id as fail_post_id
  ,ps.post_seq_id +1 as next_post_id
  FROM post_seq AS ps
  WHERE ps.is_successful_post = 0
)

SELECT
pp.user_id
,sum(p2.is_successful_post)*1.0/count(p2.is_successful_post) AS
  next_post_sc_rate
FROM post_pairings as pp
JOIN post as p2
  ON pp.next_post_id = p2.post_id
GROUP BY 1
ORDER BY next_post_sc_rate ASC
```

```
JOIN post as p2
  ON pp.next_post_id = p2.post_i
GROUP BY 1
ORDER BY next_post_sc_rate ASC
--------------

+---------+-------------------+
| user_id | next_post_sc_rate |
+---------+-------------------+
|       9 |           0.35238 |
|      13 |           0.39130 |
|      19 |           0.42105 |
|      15 |           0.42537 |
|      20 |           0.42857 |
|       6 |           0.43796 |
|       4 |           0.43902 |
|       7 |           0.44800 |
|      11 |           0.45082 |
|       8 |           0.45113 |
|      14 |           0.46552 |
|      17 |           0.47015 |
|      18 |           0.47328 |
|      10 |           0.47368 |
|       2 |           0.47788 |
|      16 |           0.50000 |
|       5 |           0.50394 |
|       3 |           0.51163 |
|      12 |           0.51220 |
|       1 |           0.51327 |
+---------+-------------------+
20 rows in set (1.89 sec)

Bye
```



Write a solution to report the customer_id and customer_name of customers who bought products **"A"**, **"B"** but did not buy the product **"C"** since we want to recommend them to purchase this product.

Return the result table **ordered** by `customer_id`.

The result format is in the following example.

Example 1:

```
Input:
Customers table:
+-------------+---------------+
| customer_id | customer_name |
+-------------+---------------+
| 1           | Daniel        |
| 2           | Diana         |
| 3           | Elizabeth     |
| 4           | Jhon          |
+-------------+---------------+
Orders table:
+------------+--------------+---------------+
| order_id   | customer_id  | product_name  |
+------------+--------------+---------------+
| 10         |     1        |     A         |
| 20         |     1        |     B         |
| 30         |     1        |     D         |
| 40         |     1        |     C         |
| 50         |     2        |     A         |
| 60         |     3        |     A         |
| 70         |     3        |     B         |
| 80         |     3        |     D         |
| 90         |     4        |     C         |
+------------+--------------+---------------+
Output:
+-------------+---------------+
| customer_id | customer_name |
+-------------+---------------+
| 3           | Elizabeth     |
```

Runtime: 874ms

Sql

```sql
# Write your MySQL query statement below
select o.customer_id, c.customer_name
from orders o
left join customers c
on o.customer_id = c.customer_id
group by customer_id
having
    sum(o.product_name = 'A') > 0 and
    sum(o.product_name = 'B') > 0 and
    sum(o.product_name = 'C') = 0
order by o.customer_id
```

Testcase | Test Result

```
| 50       | 2           | A            |
| 60       | 3           | A            |
```

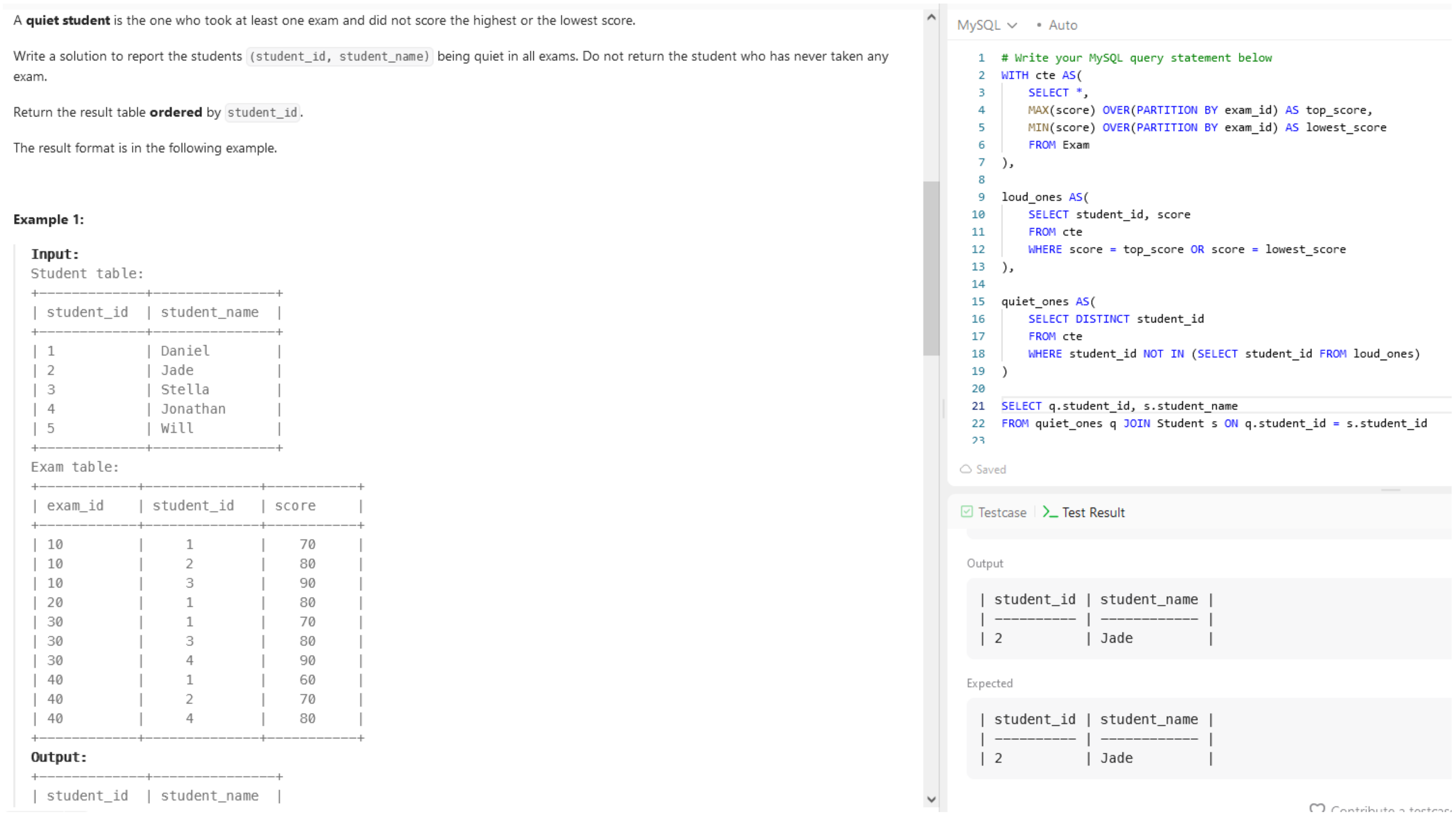
View more

Output

```
| customer_id | customer_name |
| ----------- | ------------- |
| 3           | Elizabeth     |
```

Expected

```
| customer_id | customer_name |
| ----------- | ------------- |
| 3           | Elizabeth     |
```



A **quiet student** is the one who took at least one exam and did not score the highest or the lowest score.

Write a solution to report the students `(student_id, student_name)` being quiet in all exams. Do not return the student who has never taken any exam.

Return the result table **ordered** by `student_id`.

The result format is in the following example.

Example 1:

```
Input:
Student table:
+-------------+---------------+
| student_id  | student_name  |
+-------------+---------------+
| 1           | Daniel        |
| 2           | Jade          |
| 3           | Stella        |
| 4           | Jonathan      |
| 5           | Will          |
+-------------+---------------+
Exam table:
+------------+--------------+-----------+
| exam_id    | student_id   | score     |
+------------+--------------+-----------+
| 10         |     1        |    70     |
| 10         |     2        |    80     |
| 10         |     3        |    90     |
| 20         |     1        |    80     |
| 30         |     1        |    70     |
| 30         |     3        |    80     |
| 30         |     4        |    90     |
| 40         |     1        |    60     |
| 40         |     2        |    70     |
| 40         |     4        |    80     |
+------------+--------------+-----------+
Output:
+-------------+---------------+
| student_id  | student_name  |
```

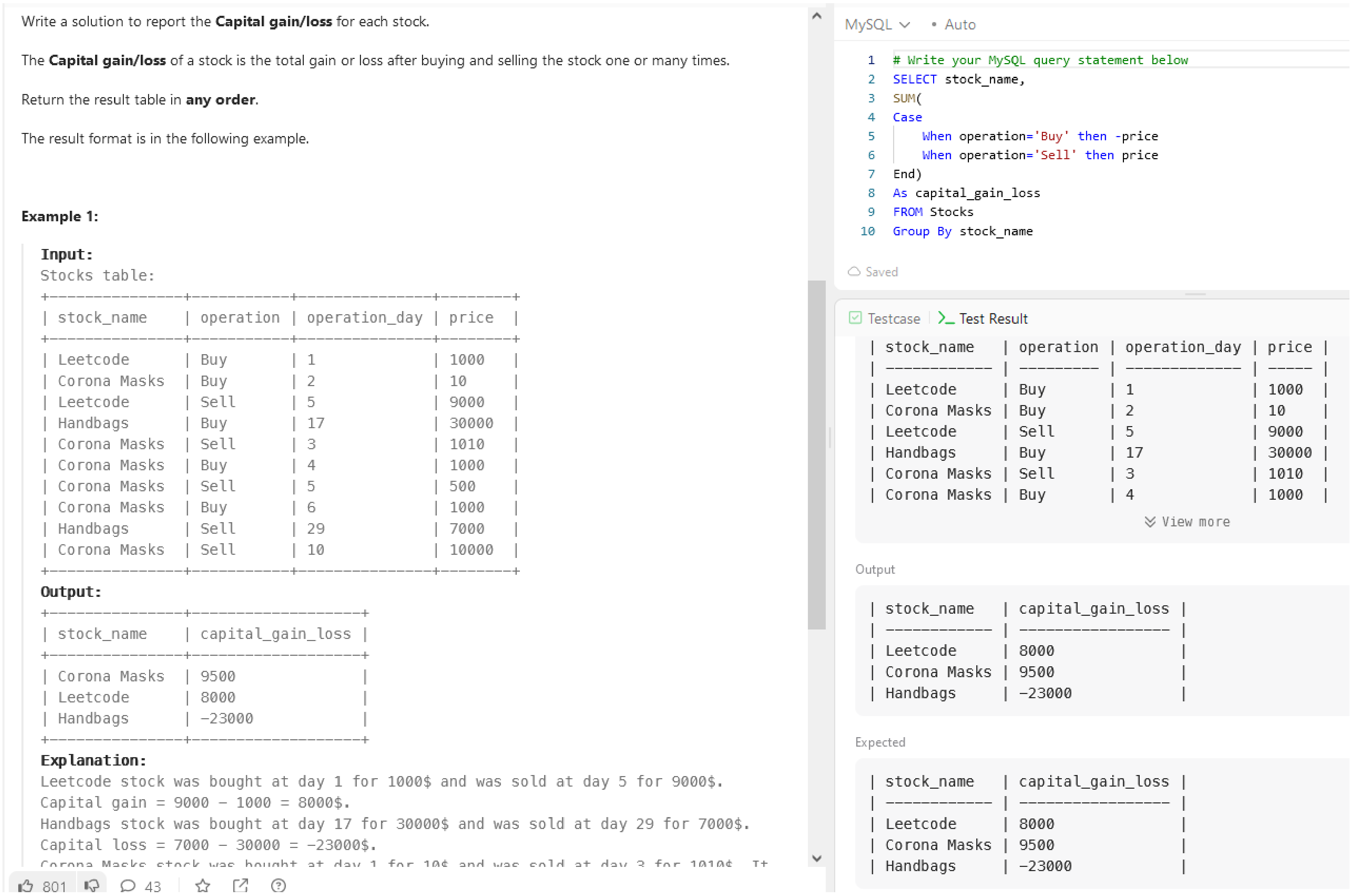
```sql
# Write your MySQL query statement below
WITH cte AS(
    SELECT *,
    MAX(score) OVER(PARTITION BY exam_id) AS top_score,
    MIN(score) OVER(PARTITION BY exam_id) AS lowest_score
    FROM Exam
),

loud_ones AS(
    SELECT student_id, score
    FROM cte
    WHERE score = top_score OR score = lowest_score
),

quiet_ones AS(
    SELECT DISTINCT student_id
    FROM cte
    WHERE student_id NOT IN (SELECT student_id FROM loud_ones)
)

SELECT q.student_id, s.student_name
FROM quiet_ones q JOIN Student s ON q.student_id = s.student_id
```

Output

```
| student_id | student_name |
| ---------- | ------------ |
| 2          | Jade         |
```

Expected

```
| student_id | student_name |
| ---------- | ------------ |
| 2          | Jade         |
```



Write a solution to report the **Capital gain/loss** for each stock.

The **Capital gain/loss** of a stock is the total gain or loss after buying and selling the stock one or many times.

Return the result table in **any order**.

The result format is in the following example.

Example 1:

Input:
```
Stocks table:
+---------------+-----------+---------------+--------+
| stock_name    | operation | operation_day | price  |
+---------------+-----------+---------------+--------+
| Leetcode      | Buy       | 1             | 1000   |
| Corona Masks  | Buy       | 2             | 10     |
| Leetcode      | Sell      | 5             | 9000   |
| Handbags      | Buy       | 17            | 30000  |
| Corona Masks  | Sell      | 3             | 1010   |
| Corona Masks  | Buy       | 4             | 1000   |
| Corona Masks  | Sell      | 5             | 500    |
| Corona Masks  | Buy       | 6             | 1000   |
| Handbags      | Sell      | 29            | 7000   |
| Corona Masks  | Sell      | 10            | 10000  |
+---------------+-----------+---------------+--------+
```
Output:
```
+---------------+-------------------+
| stock_name    | capital_gain_loss |
+---------------+-------------------+
| Corona Masks  | 9500              |
| Leetcode      | 8000              |
| Handbags      | -23000            |
+---------------+-------------------+
```
Explanation:
```
Leetcode stock was bought at day 1 for 1000$ and was sold at day 5 for 9000$.
Capital gain = 9000 - 1000 = 8000$.
Handbags stock was bought at day 17 for 30000$ and was sold at day 29 for 7000$.
Capital loss = 7000 - 30000 = -23000$.
Corona Masks stock was bought at day 1 for 10$ and was sold at day 3 for 1010$. It
```

801 43

MySQL • Auto

```sql
# Write your MySQL query statement below
SELECT stock_name,
SUM(
Case
    When operation='Buy' then -price
    When operation='Sell' then price
End)
As capital_gain_loss
FROM Stocks
Group By stock_name
```

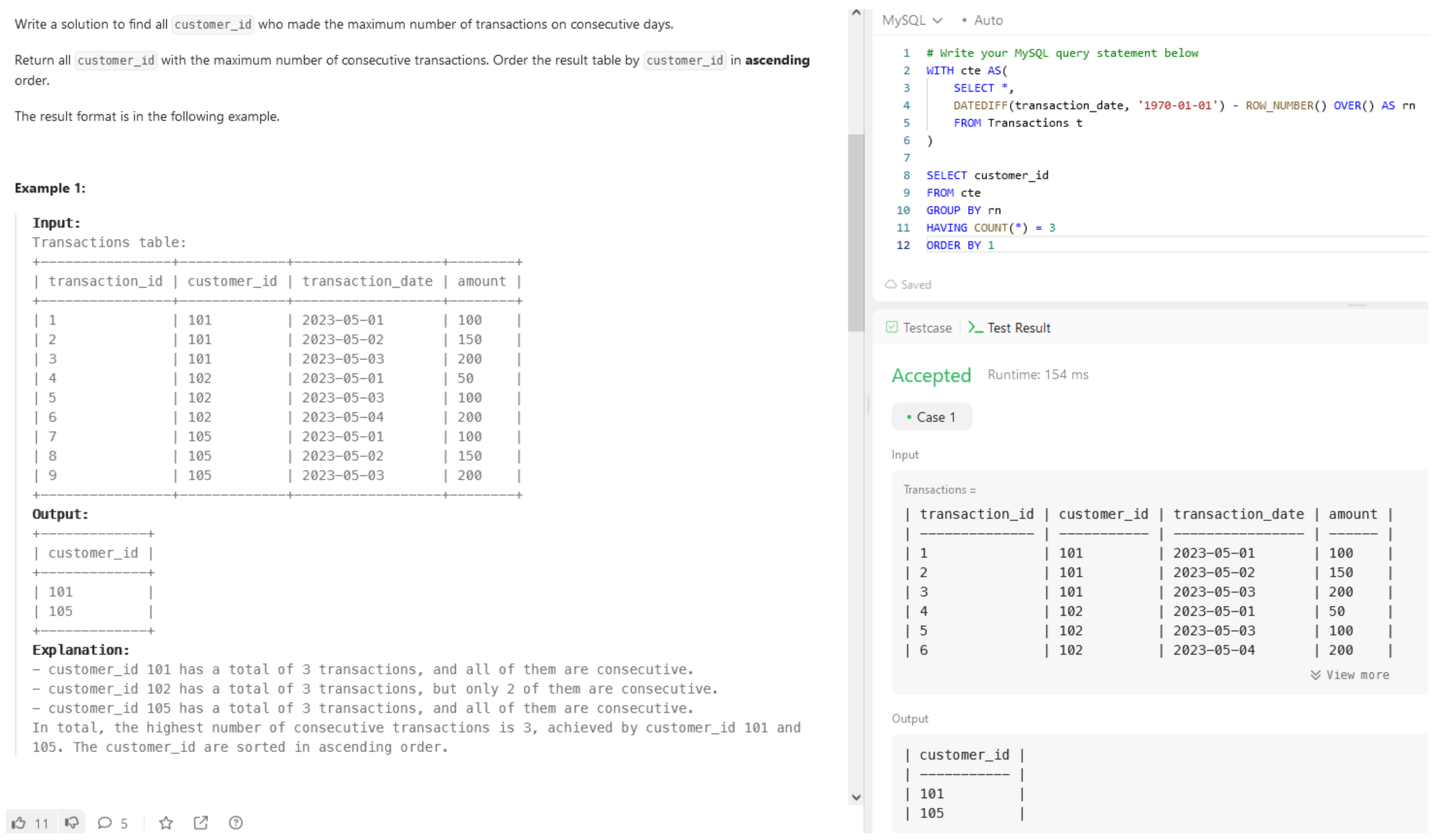
Saved

Testcase | Test Result

```
| stock_name   | operation | operation_day | price |
| ------------ | --------- | ------------- | ----- |
| Leetcode     | Buy       | 1             | 1000  |
| Corona Masks | Buy       | 2             | 10    |
| Leetcode     | Sell      | 5             | 9000  |
| Handbags     | Buy       | 17            | 30000 |
| Corona Masks | Sell      | 3             | 1010  |
| Corona Masks | Buy       | 4             | 1000  |
```
View more

Output
```
| stock_name   | capital_gain_loss |
| ------------ | ----------------- |
| Leetcode     | 8000              |
| Corona Masks | 9500              |
| Handbags     | -23000            |
```

Expected
```
| stock_name   | capital_gain_loss |
| ------------ | ----------------- |
| Leetcode     | 8000              |
| Corona Masks | 9500              |
| Handbags     | -23000            |
```



Write a solution to find all `customer_id` who made the maximum number of transactions on consecutive days.

Return all `customer_id` with the maximum number of consecutive transactions. Order the result table by `customer_id` in **ascending** order.

The result format is in the following example.

Example 1:

```
Input:
Transactions table:
+----------------+-------------+------------------+--------+
| transaction_id | customer_id | transaction_date | amount |
+----------------+-------------+------------------+--------+
| 1              | 101         | 2023-05-01       | 100    |
| 2              | 101         | 2023-05-02       | 150    |
| 3              | 101         | 2023-05-03       | 200    |
| 4              | 102         | 2023-05-01       | 50     |
| 5              | 102         | 2023-05-03       | 100    |
| 6              | 102         | 2023-05-04       | 200    |
| 7              | 105         | 2023-05-01       | 100    |
| 8              | 105         | 2023-05-02       | 150    |
| 9              | 105         | 2023-05-03       | 200    |
+----------------+-------------+------------------+--------+
Output:
+-------------+
| customer_id |
+-------------+
| 101         |
| 105         |
+-------------+
Explanation:
- customer_id 101 has a total of 3 transactions, and all of them are consecutive.
- customer_id 102 has a total of 3 transactions, but only 2 of them are consecutive.
- customer_id 105 has a total of 3 transactions, and all of them are consecutive.
In total, the highest number of consecutive transactions is 3, achieved by customer_id 101 and
105. The customer_id are sorted in ascending order.
```

```sql
# Write your MySQL query statement below
WITH cte AS(
    SELECT *,
    DATEDIFF(transaction_date, '1970-01-01') - ROW_NUMBER() OVER() AS rn
    FROM Transactions t
)

SELECT customer_id
FROM cte
GROUP BY rn
HAVING COUNT(*) = 3
ORDER BY 1
```

Accepted Runtime: 154 ms

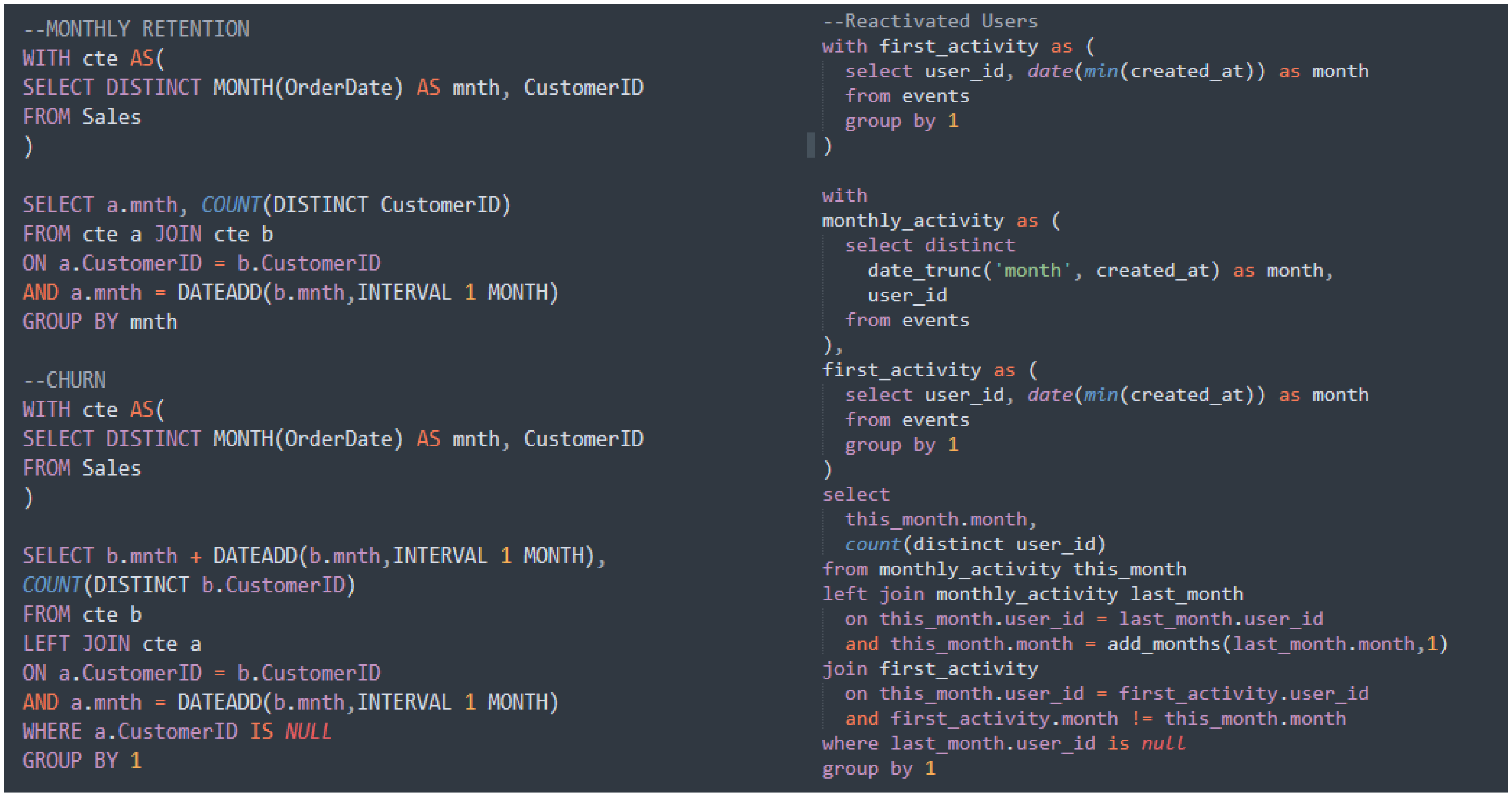
Input

Transactions =

```
| transaction_id | customer_id | transaction_date | amount |
| -------------- | ----------- | ---------------- | ------ |
| 1              | 101         | 2023-05-01       | 100    |
| 2              | 101         | 2023-05-02       | 150    |
| 3              | 101         | 2023-05-03       | 200    |
| 4              | 102         | 2023-05-01       | 50     |
| 5              | 102         | 2023-05-03       | 100    |
| 6              | 102         | 2023-05-04       | 200    |
```

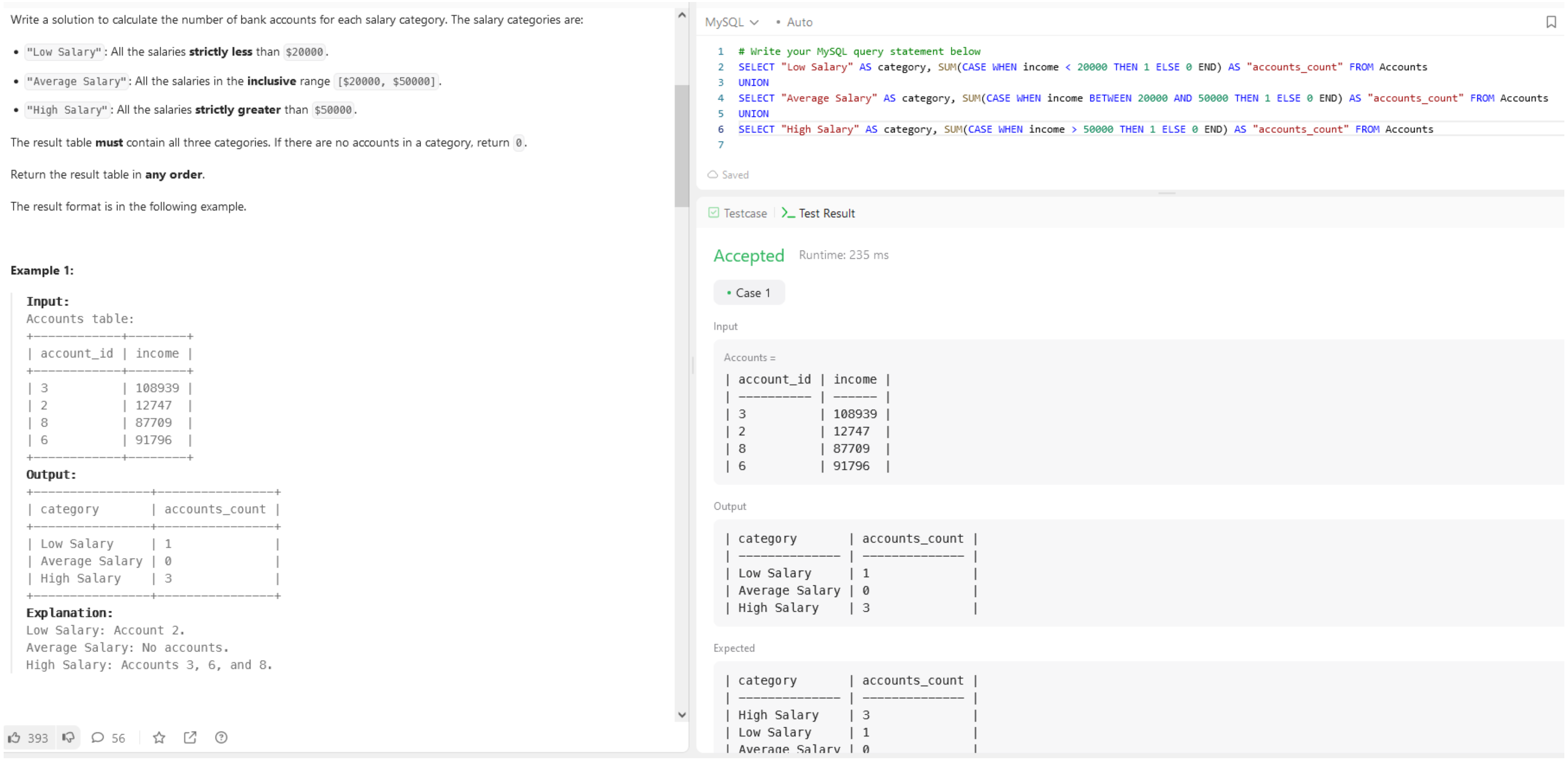
Output

```
| customer_id |
| ----------- |
| 101         |
| 105         |
```



```sql
--MONTHLY RETENTION
WITH cte AS(
SELECT DISTINCT MONTH(OrderDate) AS mnth, CustomerID
FROM Sales
)

SELECT a.mnth, COUNT(DISTINCT CustomerID)
FROM cte a JOIN cte b
ON a.CustomerID = b.CustomerID
AND a.mnth = DATEADD(b.mnth,INTERVAL 1 MONTH)
GROUP BY mnth

--CHURN
WITH cte AS(
SELECT DISTINCT MONTH(OrderDate) AS mnth, CustomerID
FROM Sales
)

SELECT b.mnth + DATEADD(b.mnth,INTERVAL 1 MONTH),
COUNT(DISTINCT b.CustomerID)
FROM cte b
LEFT JOIN cte a
ON a.CustomerID = b.CustomerID
AND a.mnth = DATEADD(b.mnth,INTERVAL 1 MONTH)
WHERE a.CustomerID IS NULL
GROUP BY 1
```

```sql
--Reactivated Users
with first_activity as (
  select user_id, date(min(created_at)) as month
  from events
  group by 1
)

with
monthly_activity as (
  select distinct
    date_trunc('month', created_at) as month,
    user_id
  from events
),
first_activity as (
  select user_id, date(min(created_at)) as month
  from events
  group by 1
)
select
  this_month.month,
  count(distinct user_id)
from monthly_activity this_month
left join monthly_activity last_month
  on this_month.user_id = last_month.user_id
  and this_month.month = add_months(last_month.month,1)
join first_activity
  on this_month.user_id = first_activity.user_id
  and first_activity.month != this_month.month
where last_month.user_id is null
group by 1
```



Write a solution to calculate the number of bank accounts for each salary category. The salary categories are:

- `"Low Salary"`: All the salaries **strictly less** than `$20000`.
- `"Average Salary"`: All the salaries in the **inclusive** range `[$20000, $50000]`.
- `"High Salary"`: All the salaries **strictly greater** than `$50000`.

The result table **must** contain all three categories. If there are no accounts in a category, return `0`.

Return the result table in **any order**.

The result format is in the following example.

Example 1:

```
Input:
Accounts table:
+------------+--------+
| account_id | income |
+------------+--------+
| 3          | 108939 |
| 2          | 12747  |
| 8          | 87709  |
| 6          | 91796  |
+------------+--------+
Output:
+----------------+----------------+
| category       | accounts_count |
+----------------+----------------+
| Low Salary     | 1              |
| Average Salary | 0              |
| High Salary    | 3              |
+----------------+----------------+
Explanation:
Low Salary: Account 2.
Average Salary: No accounts.
High Salary: Accounts 3, 6, and 8.
```

393 56

MySQL • Auto

```sql
# Write your MySQL query statement below
SELECT "Low Salary" AS category, SUM(CASE WHEN income < 20000 THEN 1 ELSE 0 END) AS "accounts_count" FROM Accounts
UNION
SELECT "Average Salary" AS category, SUM(CASE WHEN income BETWEEN 20000 AND 50000 THEN 1 ELSE 0 END) AS "accounts_count" FROM Accounts
UNION
SELECT "High Salary" AS category, SUM(CASE WHEN income > 50000 THEN 1 ELSE 0 END) AS "accounts_count" FROM Accounts

```

Saved

Testcase | Test Result

Accepted Runtime: 235 ms

• Case 1

Input

Accounts =

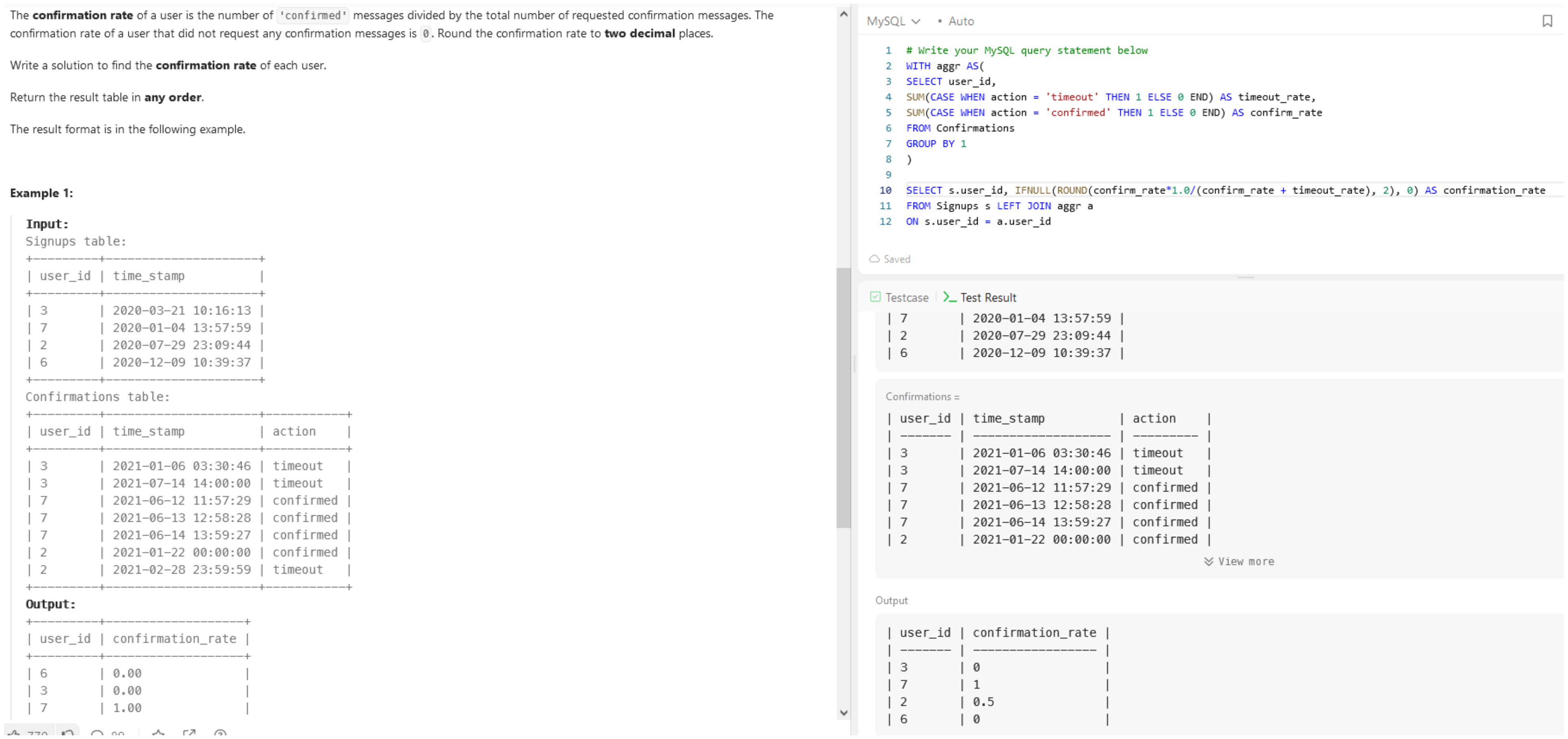
```
| account_id | income |
| ---------- | ------ |
| 3          | 108939 |
| 2          | 12747  |
| 8          | 87709  |
| 6          | 91796  |
```

Output

```
| category       | accounts_count |
| -------------- | -------------- |
| Low Salary     | 1              |
| Average Salary | 0              |
| High Salary    | 3              |
```

Expected

```
| category       | accounts_count |
| -------------- | -------------- |
| High Salary    | 3              |
| Low Salary     | 1              |
| Average Salary | 0              |
```



The **confirmation rate** of a user is the number of `'confirmed'` messages divided by the total number of requested confirmation messages. The confirmation rate of a user that did not request any confirmation messages is `0`. Round the confirmation rate to **two decimal** places.

Write a solution to find the **confirmation rate** of each user.

Return the result table in **any order**.

The result format is in the following example.

Example 1:

```
Input:
Signups table:
+---------+---------------------+
| user_id | time_stamp          |
+---------+---------------------+
| 3       | 2020-03-21 10:16:13 |
| 7       | 2020-01-04 13:57:59 |
| 2       | 2020-07-29 23:09:44 |
| 6       | 2020-12-09 10:39:37 |
+---------+---------------------+
Confirmations table:
+---------+---------------------+-----------+
| user_id | time_stamp          | action    |
+---------+---------------------+-----------+
| 3       | 2021-01-06 03:30:46 | timeout   |
| 3       | 2021-07-14 14:00:00 | timeout   |
| 7       | 2021-06-12 11:57:29 | confirmed |
| 7       | 2021-06-13 12:58:28 | confirmed |
| 7       | 2021-06-14 13:59:27 | confirmed |
| 2       | 2021-01-22 00:00:00 | confirmed |
| 2       | 2021-02-28 23:59:59 | timeout   |
+---------+---------------------+-----------+
Output:
+---------+-------------------+
| user_id | confirmation_rate |
+---------+-------------------+
| 6       | 0.00              |
| 3       | 0.00              |
| 7       | 1.00              |
```

```sql
# Write your MySQL query statement below
WITH aggr AS(
SELECT user_id,
SUM(CASE WHEN action = 'timeout' THEN 1 ELSE 0 END) AS timeout_rate,
SUM(CASE WHEN action = 'confirmed' THEN 1 ELSE 0 END) AS confirm_rate
FROM Confirmations
GROUP BY 1
)

SELECT s.user_id, IFNULL(ROUND(confirm_rate*1.0/(confirm_rate + timeout_rate), 2), 0) AS confirmation_rate
FROM Signups s LEFT JOIN aggr a
ON s.user_id = a.user_id
```

```
| 7       | 2020-01-04 13:57:59 |
| 2       | 2020-07-29 23:09:44 |
| 6       | 2020-12-09 10:39:37 |
```

Confirmations =

```
| user_id | time_stamp          | action    |
| ------- | ------------------- | --------- |
| 3       | 2021-01-06 03:30:46 | timeout   |
| 3       | 2021-07-14 14:00:00 | timeout   |
| 7       | 2021-06-12 11:57:29 | confirmed |
| 7       | 2021-06-13 12:58:28 | confirmed |
| 7       | 2021-06-14 13:59:27 | confirmed |
| 2       | 2021-01-22 00:00:00 | confirmed |
```

Output

```
| user_id | confirmation_rate |
| ------- | ----------------- |
| 3       | 0                 |
| 7       | 1                 |
| 2       | 0.5               |
| 6       | 0                 |
```

Write a solution to report the balance of each user after each transaction. You may assume that the balance of each account before any transaction is `0` and that the balance will never be below `0` at any moment.

Return the result table **in ascending order** by `account_id`, then by `day` in case of a tie.

The result format is in the following example.

Example 1:

```
Input:
Transactions table:
+------------+------------+----------+--------+
| account_id | day        | type     | amount |
+------------+------------+----------+--------+
| 1          | 2021-11-07 | Deposit  | 2000   |
| 1          | 2021-11-09 | Withdraw | 1000   |
| 1          | 2021-11-11 | Deposit  | 3000   |
| 2          | 2021-12-07 | Deposit  | 7000   |
| 2          | 2021-12-12 | Withdraw | 7000   |
+------------+------------+----------+--------+
Output:
+------------+------------+---------+
| account_id | day        | balance |
+------------+------------+---------+
| 1          | 2021-11-07 | 2000    |
| 1          | 2021-11-09 | 1000    |
| 1          | 2021-11-11 | 4000    |
| 2          | 2021-12-07 | 7000    |
| 2          | 2021-12-12 | 0       |
+------------+------------+---------+
Explanation:
Account 1:
- Initial balance is 0.
- 2021-11-07 --> deposit 2000. Balance is 0 + 2000 = 2000.
- 2021-11-09 --> withdraw 1000. Balance is 2000 - 1000 = 1000.
- 2021-11-11 --> deposit 3000. Balance is 1000 + 3000 = 4000.
Account 2:
- Initial balance is 0.
- 2021-12-07 --> deposit 7000. Balance is 0 + 7000 = 7000.
- 2021-12-12 --> withdraw 7000. Balance is 7000 - 7000 = 0.
```

MySQL • Auto

```sql
# Write your MySQL query statement below
SELECT account_id, day,
SUM(
    CASE
    WHEN type = "Deposit" THEN amount
    WHEN type = "Withdraw" THEN -amount
    END)  over (partition by account_id order by day asc) AS balance
FROM Transactions
GROUP BY 1, 2
```

Saved

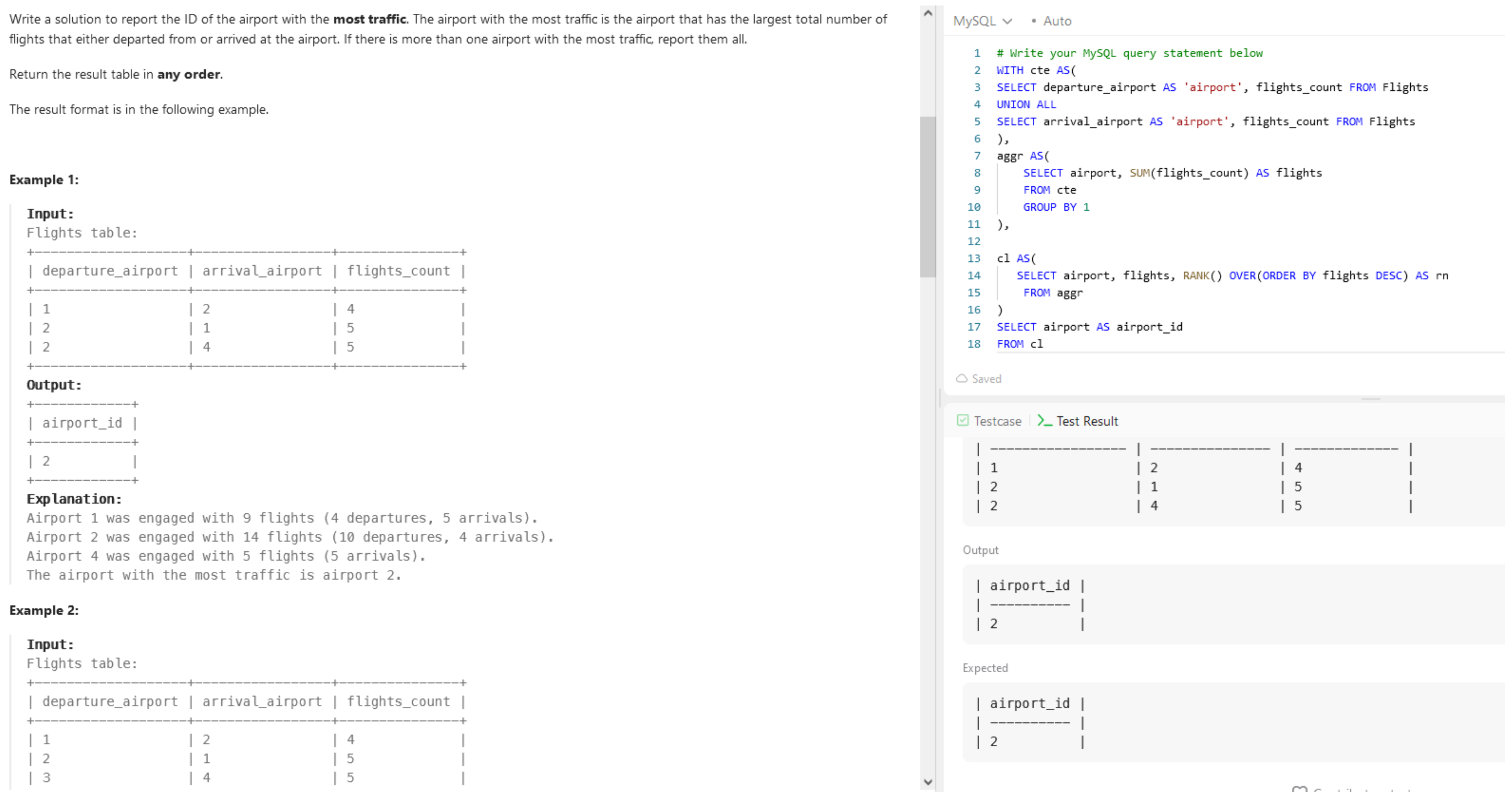
Testcase | Test Result

Output

```
| account_id | day        | balance |
| ---------- | ---------- | ------- |
| 1          | 2021-11-07 | 2000    |
| 1          | 2021-11-09 | 1000    |
| 1          | 2021-11-11 | 4000    |
| 2          | 2021-12-07 | 7000    |
| 2          | 2021-12-12 | 0       |
```

Expected

```
| account_id | day        | balance |
| ---------- | ---------- | ------- |
| 1          | 2021-11-07 | 2000    |
| 1          | 2021-11-09 | 1000    |
| 1          | 2021-11-11 | 4000    |
| 2          | 2021-12-07 | 7000    |
| 2          | 2021-12-12 | 0       |
```



Write a solution to report the ID of the airport with the **most traffic**. The airport with the most traffic is the airport that has the largest total number of flights that either departed from or arrived at the airport. If there is more than one airport with the most traffic, report them all.

Return the result table in **any order**.

The result format is in the following example.

Example 1:

```
Input:
Flights table:
+-------------------+-----------------+---------------+
| departure_airport | arrival_airport | flights_count |
+-------------------+-----------------+---------------+
| 1                 | 2               | 4             |
| 2                 | 1               | 5             |
| 2                 | 4               | 5             |
+-------------------+-----------------+---------------+
Output:
+------------+
| airport_id |
+------------+
| 2          |
+------------+
Explanation:
Airport 1 was engaged with 9 flights (4 departures, 5 arrivals).
Airport 2 was engaged with 14 flights (10 departures, 4 arrivals).
Airport 4 was engaged with 5 flights (5 arrivals).
The airport with the most traffic is airport 2.
```

Example 2:

```
Input:
Flights table:
+-------------------+-----------------+---------------+
| departure_airport | arrival_airport | flights_count |
+-------------------+-----------------+---------------+
| 1                 | 2               | 4             |
| 2                 | 1               | 5             |
| 3                 | 4               | 5             |
```

MySQL • Auto

```sql
# Write your MySQL query statement below
WITH cte AS(
SELECT departure_airport AS 'airport', flights_count FROM Flights
UNION ALL
SELECT arrival_airport AS 'airport', flights_count FROM Flights
),
aggr AS(
    SELECT airport, SUM(flights_count) AS flights
    FROM cte
    GROUP BY 1
),

cl AS(
   SELECT airport, flights, RANK() OVER(ORDER BY flights DESC) AS rn
    FROM aggr
)
SELECT airport AS airport_id
FROM cl
```

Saved

Testcase | Test Result

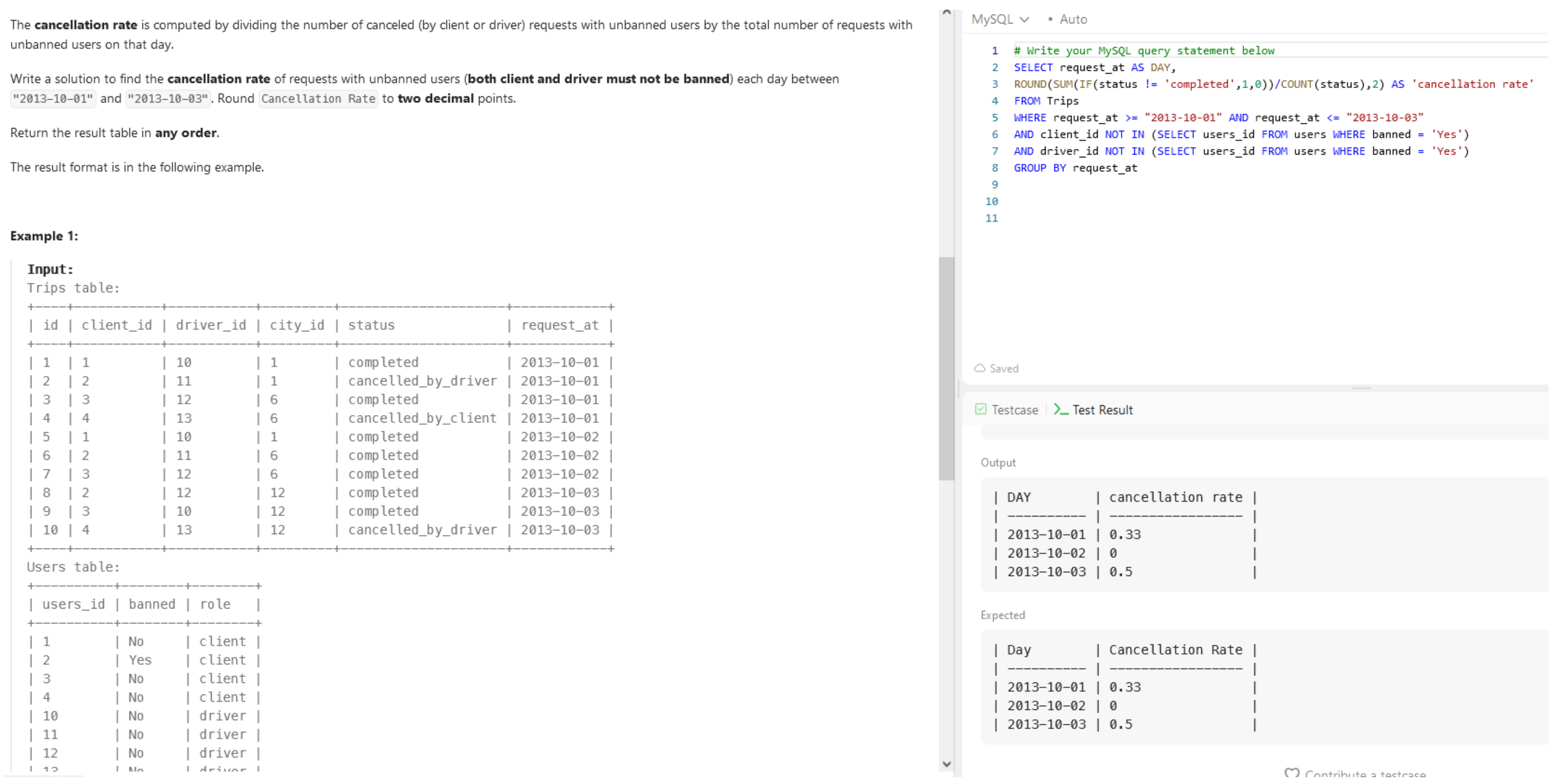
```
| ----------------- | --------------- | ------------- |
| 1                 | 2               | 4             |
| 2                 | 1               | 5             |
| 2                 | 4               | 5             |
```

Output

```
| airport_id |
| ---------- |
| 2          |
```

Expected

```
| airport_id |
| ---------- |
| 2          |
```



The **cancellation rate** is computed by dividing the number of canceled (by client or driver) requests with unbanned users by the total number of requests with unbanned users on that day.

Write a solution to find the **cancellation rate** of requests with unbanned users (**both client and driver must not be banned**) each day between `"2013-10-01"` and `"2013-10-03"`. Round `Cancellation Rate` to **two decimal** points.

Return the result table in **any order**.

The result format is in the following example.

Example 1:

```
Input:
Trips table:
+----+-----------+-----------+---------+---------------------+------------+
| id | client_id | driver_id | city_id | status              | request_at |
+----+-----------+-----------+---------+---------------------+------------+
| 1  | 1         | 10        | 1       | completed           | 2013-10-01 |
| 2  | 2         | 11        | 1       | cancelled_by_driver | 2013-10-01 |
| 3  | 3         | 12        | 6       | completed           | 2013-10-01 |
| 4  | 4         | 13        | 6       | cancelled_by_client | 2013-10-01 |
| 5  | 1         | 10        | 1       | completed           | 2013-10-02 |
| 6  | 2         | 11        | 6       | completed           | 2013-10-02 |
| 7  | 3         | 12        | 6       | completed           | 2013-10-02 |
| 8  | 2         | 12        | 12      | completed           | 2013-10-03 |
| 9  | 3         | 10        | 12      | completed           | 2013-10-03 |
| 10 | 4         | 13        | 12      | cancelled_by_driver | 2013-10-03 |
+----+-----------+-----------+---------+---------------------+------------+
Users table:
+----------+--------+--------+
| users_id | banned | role   |
+----------+--------+--------+
| 1        | No     | client |
| 2        | Yes    | client |
| 3        | No     | client |
| 4        | No     | client |
| 10       | No     | driver |
| 11       | No     | driver |
| 12       | No     | driver |
| 13       | No     | driver |
```

MySQL • Auto

```sql
# Write your MySQL query statement below
SELECT request_at AS DAY,
ROUND(SUM(IF(status != 'completed',1,0))/COUNT(status),2) AS 'cancellation rate'
FROM Trips
WHERE request_at >= "2013-10-01" AND request_at <= "2013-10-03"
AND client_id NOT IN (SELECT users_id FROM users WHERE banned = 'Yes')
AND driver_id NOT IN (SELECT users_id FROM users WHERE banned = 'Yes')
GROUP BY request_at
```

Saved

Testcase | Test Result

Output

```
| DAY        | cancellation rate |
| ---------- | ----------------- |
| 2013-10-01 | 0.33              |
| 2013-10-02 | 0                 |
| 2013-10-03 | 0.5               |
```

Expected

```
| Day        | Cancellation Rate |
| ---------- | ----------------- |
| 2013-10-01 | 0.33              |
| 2013-10-02 | 0                 |
| 2013-10-03 | 0.5               |
```

Contribute a testcase