2021/1/15

177. Nth Highest Salary

Write a SQL query to get the n^{th} highest salary from the Employee table.

```
+----+--------+
| Id | Salary |
+----+--------+
| 1  | 100    |
| 2  | 200    |
| 3  | 300    |
+----+--------+
```

For example, given the above Employee table, the n^{th} highest salary where n = 2 is 200. If there is no n^{th} highest salary, then the query should return null.

```
+------------------------+
| getNthHighestSalary(2) |
+------------------------+
| 200                    |
+------------------------+
```

```
CREATE FUNCTION getNthHighestSalary(N INT) RETURNS INT
BEGIN
DECLARE M INT;
SET M=N-1;
  RETURN (
      # Write your MySQL query statement below.
      SELECT DISTINCT Salary FROM Employee ORDER BY Salary DESC LIMIT M, 1
  );
END
```

178. Rank Scores

Write a SQL query to rank scores. If there is a tie between two scores, both should have the same ranking. Note that after a tie, the next ranking number should be the next consecutive integer value. In other words, there should be no "holes" between ranks.

```
+----+-------+
| Id | Score |
+----+-------+
| 1  | 3.50  |
| 2  | 3.65  |
| 3  | 4.00  |
| 4  | 3.85  |
| 5  | 4.00  |
| 6  | 3.65  |
+----+-------+
```

For example, given the above Scores table, your query should generate the following report (order by highest score):

```
+-------+---------+
| score | Rank    |
+-------+---------+
| 4.00  | 1       |
| 4.00  | 1       |
| 3.85  | 2       |
| 3.65  | 3       |
| 3.65  | 3       |
| 3.50  | 4       |
+-------+---------+
```

Important Note: For MySQL solutions, to escape reserved words used as column names, you can use an apostrophe before and after the keyword. For example **`Rank`**.

```
SELECT

  Score,
  (SELECT COUNT(DISTINCT Score) FROM Scores WHERE Score >=
s.Score) 'Rank'
FROM Scores s
ORDER BY Score DESC;

SELECT s.score, COUNT(DISTINCT t.score) AS 'Rank'
FROM Scores s JOIN Scores t ON s.score <= t.score
GROUP BY s.Id
ORDER BY s.score desc;
```

180. Consecutive Numbers

Table: Logs

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| num         | varchar |
+-------------+---------+
```

id is the primary key for this table.

Write an SQL query to find all numbers that appear at least three times consecutively.

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

The query result format is in the following example:

Logs table:

```
+----+-----+
| Id | Num |
+----+-----+
| 1  | 1   |
| 2  | 1   |
| 3  | 1   |
| 4  | 2   |
| 5  | 1   |
| 6  | 2   |
| 7  | 2   |
+----+-----+
```

Result table:

```
+-----------------+
| ConsecutiveNums |
+-----------------+
| 1               |
+-----------------+
```

1 is the only number that appears consecutively for at least three times.

```
SELECT T.Num AS ConsecutiveNums
FROM
(SELECT DISTINCT A.Num FROM
Logs A LEFT JOIN Logs B on A.Id = B.Id-1
        LEFT JOIN Logs C on A.Id = C.Id-2
WHERE A.Num = B.Num AND A.Num = C.Num) T;
```

184. Department Highest Salary

The Employee table holds all employees. Every employee has an Id, a salary, and there is also a column for the department Id.

```
+----+-------+--------+--------------+
| Id | Name  | Salary | DepartmentId |
+----+-------+--------+--------------+
| 1  | Joe   | 70000  | 1            |
| 2  | Jim   | 90000  | 1            |
| 3  | Henry | 80000  | 2            |
| 4  | Sam   | 60000  | 2            |
| 5  | Max   | 90000  | 1            |
+----+-------+--------+--------------+
```

The Department table holds all departments of the company.

```
+----+----------+
| Id | Name     |
+----+----------+
| 1  | IT       |
| 2  | Sales    |
+----+----------+
```

Write a SQL query to find employees who have the highest salary in each of the departments. For the above tables, your SQL query should return the following rows (order of rows does not matter).

```
+------------+----------+--------+
| Department | Employee | Salary |
+------------+----------+--------+
| IT         | Max      | 90000  |
| IT         | Jim      | 90000  |
| Sales      | Henry    | 80000  |
+------------+----------+--------+
```

Explanation:

Max and Jim both have the highest salary in the IT department and Henry has the highest salary in the Sales department.

```
SELECT
    Department.name AS 'Department',
    Employee.name AS 'Employee',
    Salary
FROM
    Employee
    JOIN
    Department ON Employee.
DepartmentId = Department.Id
WHERE
    (Employee.DepartmentId , Salary) IN
    (   SELECT
    DepartmentId, MAX(Salary)
    FROM
    Employee
    GROUP BY DepartmentId );
```

2021/1/16

534. Game Play Analysis III

Table: Activity

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| player_id    | int     |
| device_id    | int     |
| event_date   | date    |
| games_played | int     |
+--------------+---------+
```

(player_id, event_date) is the primary key of this table.

This table shows the activity of players of some game.

Each row is a record of a player who logged in and played a number of games (possibly 0) before logging out on some day using some device.

Write an SQL query that reports for each player and date, how many games played **so far** by the player. That is, the total number of games played by the player until that date. Check the example for clarity.

The query result format is in the following example:

Activity table:

```
+-----------+-----------+------------+--------------+
| player_id | device_id | event_date | games_played |
+-----------+-----------+------------+--------------+
| 1         | 2         | 2016-03-01 | 5            |
| 1         | 2         | 2016-05-02 | 6            |
| 1         | 3         | 2017-06-25 | 1            |
| 3         | 1         | 2016-03-02 | 0            |
| 3         | 4         | 2018-07-03 | 5            |
+-----------+-----------+------------+--------------+
```

Result table:

```
+-----------+------------+---------------------+
| player_id | event_date | games_played_so_far |
+-----------+------------+---------------------+
| 1         | 2016-03-01 | 5                   |
| 1         | 2016-05-02 | 11                  |
| 1         | 2017-06-25 | 12                  |
| 3         | 2016-03-02 | 0                   |
| 3         | 2018-07-03 | 5                   |
+-----------+------------+---------------------+
```

For the player with id 1, 5 + 6 = 11 games played by 2016-05-02, and 5 + 6 + 1 = 12 games played by 2017-06-25.

For the player with id 3, 0 + 5 = 5 games played by 2018-07-03.

Note that for each player we only care about the days when the player logged in.

```
SELECT a1.player_id, a1.event_date, SUM(a2.games_played) AS games_played_so_far

FROM Activity a1 JOIN Activity a2 ON a1.player_id = a2.player_id

WHERE a1.event_date >= a2.event_date

GROUP BY a1.player_id, a1.event_date

ORDER BY a1.player_id,a1.event_date;
```

550. Game Play Analysis IV

Table: Activity

```
+--------------+---------+

| Column Name  | Type    |

+--------------+---------+

| player_id    | int     |

| device_id    | int     |

| event_date   | date    |

| games_played | int     |

+--------------+---------+
```

(player_id, event_date) is the primary key of this table.

This table shows the activity of players of some game.

Each row is a record of a player who logged in and played a number of games (possibly 0) before logging out on some day using some device.

Write an SQL query that reports the **fraction** of players that logged in again on the day after the day they first logged in, **rounded to 2 decimal places**. In other words, you need to count the number of players that logged in for at least two consecutive days starting from their first login date, then divide that number by the total number of players.

The query result format is in the following example:

Activity table:

```
+-----------+-----------+------------+--------------+
| player_id | device_id | event_date | games_played |
+-----------+-----------+------------+--------------+
| 1         | 2         | 2016-03-01 | 5            |
| 1         | 2         | 2016-03-02 | 6            |
| 2         | 3         | 2017-06-25 | 1            |
| 3         | 1         | 2016-03-02 | 0            |
| 3         | 4         | 2018-07-03 | 5            |
+-----------+-----------+------------+--------------+
```

Result table:

```
+-----------+
| fraction  |
+-----------+
| 0.33      |
+-----------+
```

Only the player with id 1 logged back in after the first day he had logged in so the answer is 1/3 = 0.33

```
SELECT ROUND(COUNT(t2.player_id)/COUNT(t1.player_id),2) AS fraction
FROM
(SELECT player_id, MIN(event_date) AS first_login FROM Activity GROUP BY player_id) t1 LEFT JOIN Activity t2
ON t1.player_id = t2.player_id AND t1.first_login = t2.event_date - 1;
```

t1.first_login = t2.event_date - 1, so only those who have played right after firsts_login will be taken into account

570. Managers with at Least 5 Direct Reports

The Employee table holds all employees including their managers. Every employee has an Id, and there is also a column for the manager Id.

```
+-------+----------+-----------+----------+
|Id    |Name          |Department |ManagerId |
+-------+----------+-----------+----------+
|101   |John          |A          |null      |
|102   |Dan   |A          |101       |
|103   |James        |A         |101       |

|104   |Amy          |A         |101       |

|105   |Anne         |A         |101       |

|106   |Ron   |B         |101       |

+------+----------+-----------+----------+
```

Given the Employee table, write a SQL query that finds out managers with at least 5 direct report. For the above table, your SQL query should return:

```
+-------+

| Name  |

+-------+

| John  |

+-------+
```

Note: No one would report to himself.

```
SELECT name

FROM employee

WHERE id IN

(SELECT managerId FROM Employee

GROUP BY managerId

HAVING COUNT(managerId)>=5);
```

```
SELECT Name
FROM Employee AS t1 JOIN
(SELECT ManagerId
 FROM Employee
 GROUP BY ManagerId
 HAVING COUNT(ManagerId) >= 5) AS t2
ON t1.Id = t2.ManagerId;
```

574. Winning Candidate

Table: Candidate

```
+-----+---------+
| id  | Name    |
+-----+---------+
| 1   | A       |
| 2   | B       |
| 3   | C       |
| 4   | D       |
| 5   | E       |
+-----+---------+
```

Table: Vote

```
+-----+--------------+

| id  | CandidateId  |

+-----+--------------+

| 1   |     2       |

| 2   |     4       |

| 3   |     3       |

| 4   |     2       |

| 5   |     5       |

+-----+--------------+
```

id is the auto-increment primary key,

CandidateId is the id appeared in Candidate table.

Write a sql to find the name of the winning candidate, the above example will return the winner B.

```
+------+

| Name |

+------+

| B    |

+------+
```

Notes: You may assume **there is no tie**, in other words there will be **only one** winning candidate.

```
SELECT name AS Name

FROM Candidate JOIN


(SELECT Candidateid

 FROM Vote

 GROUP BY Candidateid

 ORDER BY COUNT(*) DESC LIMIT 1) AS winner


WHERE Candidate.id = winner.Candidateid;
```

578. Get Highest Answer Rate Question

Get the highest answer rate question from a table survey_log with these columns: **id**, **action**, **question_id**, **answer_id**, **q_num**, **timestamp**.

id means user id; action has these kind of values: "show", "answer", "skip"; answer_id is not null when action column is "answer", while is null for "show" and "skip"; q_num is the numeral order of the question in current session.

Write a sql query to identify the question which has the highest answer rate.

Example:

Input:

```
+-------+-----------+--------------+------------+-----------+------------+

| id    | action    | question_id  | answer_id  | q_num     | timestamp  |
+------+-----------+--------------+------------+-----------+------------+
| 5    | show      | 285          | null       | 1         | 123        |
| 5    | answer    | 285          | 124124     | 1         | 124        |
| 5    | show      | 369          | null       | 2         | 125        |
| 5    | skip      | 369          | null       | 2         | 126        |
+------+-----------+--------------+------------+-----------+------------+
```

Output:

```
+-------------+
| survey_log  |
+-------------+
|    285      |
+-------------+
```

Explanation:

question 285 has answer rate 1/1, while question 369 has 0/1 answer rate, so output 285.

```
SELECT question_id AS 'survey_log'

FROM survey_log

GROUP BY question_id

ORDER BY COUNT(answer_id) / SUM(IF(action = 'show', 1, 0)) DESC

LIMIT 1;
```

580. Count Student Number in Departments

A university uses 2 data tables, ***student*** and ***department***, to store data about its students and the departments associated with each major.

Write a query to print the respective department name and number of students majoring in each department for all departments in the ***department*** table (even ones with no current students).

Sort your results by descending number of students; if two or more departments have the same number of students, then sort those departments alphabetically by department name.

The ***student*** is described as follow:

Column Name	Type
student_id	Integer
student_name	String
gender	Character
dept_id	Integer

where student_id is the student's ID number, student_name is the student's name, gender is their gender, and dept_id is the department ID associated with their declared major.

And the ***department*** table is described as below:

Column Name	Type
dept_id	Integer

dept_name	String

where dept_id is the department's ID number and dept_name is the department name.

Here is an example **input**:

student table:

student_id	student_name	gender	dept_id
1	Jack	M	1
2	Jane	F	1
3	Mark	M	2

department table:

dept_id	dept_name
1	Engineering
2	Science
3	Law

The **Output** should be:

dept_name	student_number

| Engineering | 2 |

| Science | 1 |

| Law | 0 |

```
SELECT dept_name, COUNT(student_id) AS student_number
FROM department d LEFT JOIN student s ON d.dept_id = s.dept_id
GROUP BY dept_name
ORDER BY student_number DESC, dept_name ASC;
```

2021/1/17

585. Investments in 2016

Write a query to print the sum of all total investment values in 2016 (**TIV_2016**), to a scale of 2 decimal places, for all policy holders who meet the following criteria:

1. Have the same **TIV_2015** value as one or more other policyholders.
2. Are not located in the same city as any other policyholder (i.e.: the (latitude, longitude) attribute pairs must be unique).

Input Format:

The ***insurance*** table is described as follows:

Column Name	Type
PID	INTEGER(11)
TIV_2015	NUMERIC(15,2)

TIV_2016	NUMERIC(15,2)
LAT	NUMERIC(5,2)
LON	NUMERIC(5,2)

where **PID** is the policyholder's policy ID, **TIV_2015** is the total investment value in 2015, **TIV_2016** is the total investment value in 2016, **LAT** is the latitude of the policy holder's city, and **LON** is the longitude of the policy holder's city.

Sample Input

PID	TIV_2015	TIV_2016	LAT	LON
1	10	5	10	10
2	20	20	20	20
3	10	30	20	20
4	10	40	40	40

Sample Output

TIV_2016
45.00

Explanation

The first record in the table, like the last record, meets both of the two criteria.

The **TIV_2015** value '10' is as the same as the third and forth record, and its location unique.

The second record does not meet any of the two criteria. Its **TIV_2015** is not like any other policyholders.

And its location is the same with the third record, which makes the third record fail, too.

So, the result is the sum of **TIV_2016** of the first and last record, which is 45.

```
SELECT ROUND(SUM(TIV_2016), 2) AS TIV_2016

FROM insurance

WHERE

PID IN (SELECT PID FROM insurance GROUP BY LAT, LON HAVING COUNT(*) = 1)

AND

PID NOT IN

(SELECT PID FROM insurance GROUP BY TIV_2015 HAVING COUNT(*) = 1);
```

602. Friend Requests II: Who Has the Most Friends

In social network like Facebook or Twitter, people send friend requests and accept others' requests as well.

Table request_accepted

```
+--------------+-------------+------------+
| requester_id | accepter_id | accept_date|
|--------------|-------------|------------|
| 1            | 2           | 2016_06-03 |
| 1            | 3           | 2016-06-08 |
| 2            | 3           | 2016-06-08 |
| 3            | 4           | 2016-06-09 |
+--------------+-------------+------------+
```

This table holds the data of friend acceptance, while **requester_id** and **accepter_id** both are the id of a person.

Write a query to find the the people who has most friends and the most friends number under the following rules:

- It is guaranteed there is only 1 people having the most friends.
- The friend request could only been accepted once, which mean there is no multiple records with the same **requester_id** and **accepter_id** value.

For the sample data above, the result is:

Result table:

```
+------+------+
| id   | num  |
|------|------|
| 3    | 3    |
+------+------+
```

The person with id '3' is a friend of people '1', '2' and '4', so he has 3 friends in total, which is the most number than any others.

```
SELECT id, COUNT(*) AS num
FROM
((SELECT requester_id AS id FROM request_accepted)
  UNION ALL
 (SELECT accepter_id AS id FROM request_accepted)) AS tb
GROUP BY id
ORDER BY num DESC LIMIT 1;
```

608. Tree Node

Given a table tree, **id** is identifier of the tree node and **p_id** is its parent node's **id**.

```
+----+------+
| id | p_id |
+----+------+
| 1  | null |
| 2  | 1    |
| 3  | 1    |
| 4  | 2    |

| 5  | 2    |

+----+------+
```

Each node in the tree can be one of three types:

- Leaf: if the node is a leaf node.
- Root: if the node is the root of the tree.
- Inner: If the node is neither a leaf node nor a root node.

Write a query to print the node id and the type of the node. Sort your output by the node id. The result for the above sample is:

```
+----+------+

| id | Type |

+----+------+

| 1  | Root |

| 2  | Inner|

| 3  | Leaf |

| 4  | Leaf |

| 5  | Leaf |

+----+------+
```

Explanation

- Node '1' is root node, because its parent node is NULL and it has child node '2' and '3'.
- Node '2' is inner node, because it has parent node '1' and child node '4' and '5'.
- Node '3', '4' and '5' is Leaf node, because they have parent node and they don't have child node.

And here is the image of the sample tree as below:

```
        1

      /   \

    2       3

  /   \

4       5
```

-

  Note

  If there is only one node on the tree, you only need to output its root attributes.

```
SELECT DISTINCT t1.id, (
        CASE WHEN t1.p_id IS NULL THEN 'Root'
        WHEN t1.p_id IS NOT NULL AND t2.id IS NOT NULL THEN 'Inner'
        WHEN t1.p_id IS NOT NULL AND t2.id IS NULL THEN 'Leaf'
        END) AS Type
FROM tree t1 LEFT JOIN tree t2 ON t1.id = t2.p_id;
```

612. Shortest Distance in a Plane

Table point_2d holds the coordinates (x,y) of some unique points (more than two) in a plane.

Write a query to find the shortest distance between these points rounded to 2 decimals.

x	y
-1	-1
0	0
-1	-2

The shortest distance is 1.00 from point (-1,-1) to (-1,2). So the output should be:

```
| shortest |
|----------|
| 1.00     |
```

Note: The longest distance among all the points are less than 10000.

```
SELECT ROUND(SQRT(MIN((POW(p1.x - p2.x, 2) + POW(p1.y - p2.y, 2)))), 2) AS shortest
FROM point_2d p1 JOIN point_2d p2 ON p1.x != p2.x OR p1.y != p2.y;
```

614. Second Degree Follower

In facebook, there is a follow table with two columns: **followee**, **follower**.

Please write a sql query to get the amount of each follower’s follower if he/she has one.

For example:

```
+-------------+------------+
| followee    | follower   |
+-------------+------------+
|     A       |     B      |
|     B       |     C      |
|     B       |     D      |
|     D       |     E      |
+-------------+------------+
```

should output:

```
+-------------+------------+
| follower    | num        |
+-------------+------------+
|     B       |  2         |
|     D       |  1         |
+-------------+------------+
```

Explaination:

Both B and D exist in the follower list, when as a followee, B's follower is C and D, and D's follower is E. A does not exist in follower list.

Note:

Followee would not follow himself/herself in all cases.

Please display the result in follower's alphabet order.

```
SELECT f1.follower, COUNT(DISTINCT f2.follower) as num
FROM follow f1 JOIN follow f2 ON f1.follower = f2.followee
GROUP BY f1.follower
ORDER BY f1.follower;
```

626. Exchange Seats

Mary is a teacher in a middle school and she has a table seat storing students' names and their corresponding seat ids.

The column **id** is continuous increment.

Mary wants to change seats for the adjacent students.

Can you write a SQL query to output the result for Mary?

```
+---------+---------+

|    id   | student |

+---------+---------+

|    1    | Abbot   |

|    2    | Doris   |

|    3    | Emerson |

|    4    | Green   |

|    5    | Jeames  |

+---------+---------+
```

For the sample input, the output is:

```
+---------+---------+

|    id   | student |

+---------+---------+

|    1    | Doris   |
|    2    | Abbot   |
|    3    | Green   |
|    4    | Emerson |
|    5    | Jeames  |
+---------+---------+
```

Note: If the number of students is odd, there is no need to change the last one's seat.

```
SELECT CASE WHEN id % 2 = 0 THEN id - 1
        WHEN id % 2 = 1 AND id < (SELECT COUNT(*) FROM seat) THEN id + 1
        ELSE id
        END
        AS id,
student
FROM seat
ORDER BY id;
```

1045. Customers Who Bought All Products

Table: Customer

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| customer_id | int     |
| product_key | int     |
+-------------+---------+
```

product_key is a foreign key to Product table.

Table: Product

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| product_key | int     |
+-------------+---------+
```

product_key is the primary key column for this table.

Write an SQL query for a report that provides the customer ids from the Customer table that bought all the products in the Product table.

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

The query result format is in the following example:

Customer table:

```
+-------------+-------------+
| customer_id | product_key |
+-------------+-------------+
| 1           | 5           |
| 2           | 6           |
| 3           | 5           |
| 3           | 6           |
| 1           | 6           |
+-------------+-------------+
```

Product table:

```
+-------------+
| product_key |
+-------------+
| 5           |
| 6           |
+-------------+
```

Result table:

```
+-------------+
| customer_id |
+-------------+
| 1           |
| 3           |
+-------------+
```

The customers who bought all the products (5 and 6) are customers with id 1 and 3.

```
SELECT customer_id
FROM customer c
GROUP BY customer_id
HAVING COUNT(DISTINCT product_key) = (SELECT COUNT(DISTINCT product_key)
                                       FROM product);
```

1070. Product Sales Analysis III

Table: Sales

```
+-------------+-------+
| Column Name | Type  |
+-------------+-------+
| sale_id     | int   |
| product_id  | int   |
| year        | int   |
| quantity    | int   |
| price       | int   |
+-------------+-------+
```

sale_id is the primary key of this table.

product_id is a foreign key to Product table.

Note that the price is per unit.

Table: Product

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| product_id   | int     |
| product_name | varchar |
+--------------+---------+
```

product_id is the primary key of this table.

Write an SQL query that selects the **product id**, **year**, **quantity**, and **price** for the **first year** of every product sold.

The query result format is in the following example:

Sales table:

```
+---------+------------+------+----------+-------+

| sale_id | product_id | year | quantity | price |

+---------+------------+------+----------+-------+

| 1       | 100        | 2008 | 10       | 5000  |

| 2       | 100        | 2009 | 12       | 5000  |

| 7       | 200        | 2011 | 15       | 9000  |

+---------+------------+------+----------+-------+
```

Product table:

```
+------------+--------------+

| product_id | product_name |

+------------+--------------+

| 100        | Nokia        |

| 200        | Apple        |
| 300        | Samsung      |
+------------+--------------+
```

Result table:

```
+------------+------------+----------+-------+
| product_id | first_year | quantity | price |
+------------+------------+----------+-------+
| 100        | 2008       | 10       | 5000  |
| 200        | 2011       | 15       | 9000  |
+------------+------------+----------+-------+
```

```
SELECT product_id, year AS first_year, quantity, price
FROM Sales
WHERE (product_id, year)

IN

(SELECT product_id, MIN(year) AS min_year_product
FROM Sales
GROUP BY product_id) ;
```

1077. Project Employees III

Table: Project

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| project_id  | int     |
| employee_id | int     |
+-------------+---------+
```

(project_id, employee_id) is the primary key of this table.

employee_id is a foreign key to Employee table.

Table: Employee

```
+------------------+---------+
| Column Name      | Type    |
+------------------+---------+
| employee_id      | int     |
| name             | varchar |
| experience_years | int     |
+------------------+---------+
```

employee_id is the primary key of this table.

Write an SQL query that reports the **most experienced** employees in each project. In case of a tie, report all employees with the maximum number of experience years.

The query result format is in the following example:

Project table:

```
+-------------+-------------+
| project_id  | employee_id |
+-------------+-------------+
| 1           | 1           |
| 1           | 2           |
| 1           | 3           |
| 2           | 1           |
| 2           | 4           |
+-------------+-------------+
```

Employee table:

```
+-------------+--------+------------------+
| employee_id | name   | experience_years |
+-------------+--------+------------------+
| 1           | Khaled | 3                |
| 2           | Ali    | 2                |
| 3           | John   | 3                |
| 4           | Doe    | 2                |
+-------------+--------+------------------+
```

Result table:

```
+-------------+---------------+
| project_id  | employee_id   |
+-------------+---------------+
| 1           | 1             |
| 1           | 3             |
| 2           | 1             |
+-------------+---------------+
```

Both employees with id 1 and 3 have the most experience among the employees of the first project. For the second project, the employee with id 1 has the most experience.

```
SELECT p.project_id, p.employee_id
FROM Project p JOIN Employee e ON p.employee_id = e.employee_id
WHERE (p.project_id, e.experience_years) IN
```

```
(SELECT a.project_id, MAX(b.experience_years)
 FROM Project a JOIN Employee b
 ON a.employee_id = b.employee_id
 GROUP BY a.project_id);
```

```
SELECT p.project_id, e.employee_id
FROM project p INNER JOIN employee e ON e.employee_id =
p.employee_id
WHERE (p.project_id, e.experience_years) IN
(SELECT p.project_id, MAX(e.experience_years)
 FROM project p INNER JOIN employee e on e.employee_id =
p.employee_id
 GROUP BY project_id);
```

2021/1/18

1098. Unpopular Books

Table: Books

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| book_id        | int     |
| name           | varchar |
| available_from | date    |
+----------------+---------+
```

book_id is the primary key of this table.

Table: Orders

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| order_id       | int     |
| book_id        | int     |
| quantity       | int     |
| dispatch_date  | date    |
+----------------+---------+
```

order_id is the primary key of this table.

book_id is a foreign key to the Books table.

Write an SQL query that reports the **books** that have sold **less than 10** copies in the last year, excluding books that have been available for less than 1 month from today. **Assume today is 2019-06-23**.

The query result format is in the following example:

Books table:

```
+---------+--------------------+----------------+
| book_id | name               | available_from |
+---------+--------------------+----------------+
| 1       | "Kalila And Demna" | 2010-01-01     |
| 2       | "28 Letters"       | 2012-05-12     |
| 3       | "The Hobbit"       | 2019-06-10     |
| 4       | "13 Reasons Why"   | 2019-06-01     |
| 5       | "The Hunger Games" | 2008-09-21     |
+---------+--------------------+----------------+
```

Orders table:

```
+----------+---------+----------+---------------+
| order_id | book_id | quantity | dispatch_date |
+----------+---------+----------+---------------+
| 1        | 1       | 2        | 2018-07-26    |
| 2        | 1       | 1        | 2018-11-05    |
| 3        | 3       | 8        | 2019-06-11    |
| 4        | 4       | 6        | 2019-06-05    |
| 5        | 4       | 5        | 2019-06-20    |
| 6        | 5       | 9        | 2009-02-02    |
| 7        | 5       | 8        | 2010-04-13    |
+----------+---------+----------+---------------+
```

Result table:

```
+-----------+--------------------+
| book_id   | name               |
+-----------+--------------------+
| 1         | "Kalila And Demna" |
| 2         | "28 Letters"       |
| 5         | "The Hunger Games" |
+-----------+--------------------+
```

```
SELECT b.book_id, b.name
FROM books b LEFT JOIN

(SELECT book_id, SUM(quantity) AS book_sold
FROM Orders
WHERE dispatch_date BETWEEN '2018-06-23' AND '2019-06-23'
GROUP BY book_id) AS t

ON b.book_id = t.book_id
WHERE (book_sold IS NULL OR book_sold < 10) AND available_from <
'2019-05-23'
```

```
ORDER BY book_id;
```

1107. New Users Daily Count

Table: Traffic

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| activity      | enum    |
| activity_date | date    |
+---------------+---------+
```

There is no primary key for this table, it may have duplicate rows.

The activity column is an ENUM type of ('login', 'logout', 'jobs', 'groups', 'homepage').

Write an SQL query that reports for every date within at most **90 days** from today, the number of users that logged in for the first time on that date. Assume today is **2019-06-30**.

The query result format is in the following example:

Traffic table:

```
+---------+----------+---------------+
| user_id | activity | activity_date |
+---------+----------+---------------+
| 1       | login    | 2019-05-01    |
| 1       | homepage | 2019-05-01    |
| 1       | logout   | 2019-05-01    |
| 2       | login    | 2019-06-21    |
| 2       | logout   | 2019-06-21    |
| 3       | login    | 2019-01-01    |
| 3       | jobs     | 2019-01-01    |
| 3       | logout   | 2019-01-01    |
| 4       | login    | 2019-06-21    |
| 4       | groups   | 2019-06-21    |
| 4       | logout   | 2019-06-21    |
| 5       | login    | 2019-03-01    |
| 5       | logout   | 2019-03-01    |
| 5       | login    | 2019-06-21    |
| 5       | logout   | 2019-06-21    |
+---------+----------+---------------+
```

Result table:

```
+------------+-------------+
| login_date | user_count  |
+------------+-------------+
| 2019-05-01 | 1           |
| 2019-06-21 | 2           |
+------------+-------------+
```

Note that we only care about dates with non zero user count.

The user with id 5 first logged in on 2019-03-01 so he's not counted on 2019-06-21.

```
SELECT login_date, COUNT(user_id) AS user_count
FROM

(SELECT user_id, MIN(activity_date) AS login_date
FROM Traffic
WHERE activity = 'login'
GROUP BY user_id) AS t

WHERE DATEDIFF('2019-06-30', login_date) <= 90
GROUP BY login_date;
```

1112. Highest Grade For Each Student

Table: Enrollments

```
+---------------+---------+

| Column Name   | Type    |

+---------------+---------+

| student_id    | int     |

| course_id     | int     |

| grade         | int     |

+---------------+---------+
```

(student_id, course_id) is the primary key of this table.

Write a SQL query to find the highest grade with its corresponding course for each student. In case of a tie, you should find the course with the smallest course_id. The output must be sorted by increasing student_id.

The query result format is in the following example:

Enrollments table:

```
+------------+-------------------+

| student_id | course_id | grade |

+------------+-----------+-------+

| 2          | 2         | 95    |

| 2          | 3         | 95    |
| 1          | 1         | 90    |
| 1          | 2         | 99    |
| 3          | 1         | 80    |
| 3          | 2         | 75    |
| 3          | 3         | 82    |
+------------+-----------+-------+
```

Result table:

```
+------------+-------------------+
| student_id | course_id | grade |
+------------+-----------+-------+
| 1          | 2         | 99    |
| 2          | 2         | 95    |
| 3          | 3         | 82    |
+------------+-----------+-------+
```

```
SELECT DISTINCT student_id,MIN(course_id) AS course_id,grade
FROM Enrollments
WHERE (student_id,grade) IN

(SELECT DISTINCT student_id, max(grade)
 FROM Enrollments
 GROUP BY student_id)


GROUP BY student_id

ORDER BY student_id;
```

1126. Active Businesses

Table: Events

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| business_id   | int     |
| event_type    | varchar |
| occurences    | int     |
+---------------+---------+
```

(business_id, event_type) is the primary key of this table.

Each row in the table logs the info that an event of some type occured at some business for a number of times.

Write an SQL query to find all *active businesses*.

An active business is a business that has more than one event type with occurences greater than the average occurences of that event type among all businesses.

The query result format is in the following example:

Events table:

```
+-------------+------------+------------+
| business_id | event_type | occurences |
+-------------+------------+------------+
| 1           | reviews    | 7          |
| 3           | reviews    | 3          |
| 1           | ads        | 11         |
| 2           | ads        | 7          |
| 3           | ads        | 6          |
| 1           | page views | 3          |
| 2           | page views | 12         |
+-------------+------------+------------+
```

Result table:

```
+-------------+
| business_id |
+-------------+
| 1           |
+-------------+
```

Average for 'reviews', 'ads' and 'page views' are (7+3)/2=5, (11+7+6)/3=8, (3+12)/2=7.5 respectively.

Business with id 1 has 7 'reviews' events (more than 5) and 11 'ads' events (more than 8) so it is an active business.

```
SELECT business_id

FROM


(SELECT event_type, AVG(occurences) AS avg_occurences

FROM Events e1

GROUP BY event_type) AS t


JOIN Events e2 ON e2.event_type = t.event_type

WHERE e2.occurences > t.avg_occurences

GROUP BY business_id

HAVING COUNT(DISTINCT t.event_type) > 1;
```

1132. Reported Posts II

Table: Actions

```
+---------------+---------+

| Column Name   | Type    |

+---------------+---------+

| user_id       | int     |

| post_id       | int     |
| action_date   | date    |
| action        | enum    |
| extra         | varchar |
+---------------+---------+
```

There is no primary key for this table, it may have duplicate rows.

The action column is an ENUM type of ('view', 'like', 'reaction', 'comment', 'report', 'share').

The extra column has optional information about the action such as a reason for report or a type of reaction.

Table: Removals

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| post_id       | int     |
| remove_date   | date    |
+---------------+---------+
```

post_id is the primary key of this table.

Each row in this table indicates that some post was removed as a result of being reported or as a result of an admin review.

Write an SQL query to find the average for daily percentage of posts that got removed after being reported as spam, **rounded to 2 decimal places**.

The query result format is in the following example:

Actions table:

```
+---------+---------+-------------+--------+--------+
| user_id | post_id | action_date | action | extra  |
+---------+---------+-------------+--------+--------+
| 1       | 1       | 2019-07-01  | view   | null   |
| 1       | 1       | 2019-07-01  | like   | null   |
| 1       | 1       | 2019-07-01  | share  | null   |
| 2       | 2       | 2019-07-04  | view   | null   |
| 2       | 2       | 2019-07-04  | report | spam   |
| 3       | 4       | 2019-07-04  | view   | null   |
| 3       | 4       | 2019-07-04  | report | spam   |
| 4       | 3       | 2019-07-02  | view   | null   |
| 4       | 3       | 2019-07-02  | report | spam   |
| 5       | 2       | 2019-07-03  | view   | null   |
| 5       | 2       | 2019-07-03  | report | racism |
| 5       | 5       | 2019-07-03  | view   | null   |
| 5       | 5       | 2019-07-03  | report | racism |
+---------+---------+-------------+--------+--------+
```

Removals table:

```
+---------+-------------+
| post_id | remove_date |
+---------+-------------+
| 2       | 2019-07-20  |
| 3       | 2019-07-18  |
+---------+-------------+
```

Result table:

```
+-----------------------+
| average_daily_percent |
+-----------------------+
| 75.00                 |
+-----------------------+
```

The percentage for 2019-07-04 is 50% because only one post of two spam reported posts was removed.

The percentage for 2019-07-02 is 100% because one post was reported as spam and it was removed.

The other days had no spam reports so the average is (50 + 100) / 2 = 75%

Note that the output is only one number and that we do not care about the remove dates.

```
SELECT round(SUM(percent)/count(DISTINCT action_date),2) AS average_daily_percent

FROM


(SELECT a.action_date,COUNT(DISTINCT r.post_id)/ COUNT(DISTINCT a.post_id)*100 AS percent

 FROM actions a LEFT JOIN removals r ON a.post_id = r.post_id

 WHERE a.extra='spam'

 GROUP BY action_date) temp;
```

1149. Article Views II

Table: Views

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| article_id    | int     |
| author_id     | int     |
| viewer_id     | int     |
| view_date     | date    |
+---------------+---------+
```

There is no primary key for this table, it may have duplicate rows.

Each row of this table indicates that some viewer viewed an article (written by some author) on some date.

Note that equal author_id and viewer_id indicate the same person.

Write an SQL query to find all the people who viewed more than one article on the same date, sorted in ascending order by their id.

The query result format is in the following example:

Views table:

```
+------------+-----------+-----------+------------+
| article_id | author_id | viewer_id | view_date  |
+------------+-----------+-----------+------------+
| 1          | 3         | 5         | 2019-08-01 |
| 3          | 4         | 5         | 2019-08-01 |
| 1          | 3         | 6         | 2019-08-02 |
| 2          | 7         | 7         | 2019-08-01 |
| 2          | 7         | 6         | 2019-08-02 |
| 4          | 7         | 1         | 2019-07-22 |
| 3          | 4         | 4         | 2019-07-21 |
| 3          | 4         | 4         | 2019-07-21 |
+------------+-----------+-----------+------------+
```

Result table:

```
+------+
| id   |
+------+
| 5    |
| 6    |
+------+
```

```
SELECT DISTINCT viewer_id AS id
FROM Views
GROUP BY viewer_id, view_date
HAVING COUNT(DISTINCT article_id) > 1
ORDER BY id ASC;
```

1158. Market Analysis I

Table: Users

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| user_id        | int     |
| join_date      | date    |
| favorite_brand | varchar |
+----------------+---------+
```

user_id is the primary key of this table.

This table has the info of the users of an online shopping website where users can sell and buy items.

Table: Orders

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| order_id      | int     |
| order_date    | date    |
| item_id       | int     |
| buyer_id      | int     |
| seller_id     | int     |
+---------------+---------+
```

order_id is the primary key of this table.

item_id is a foreign key to the Items table.

buyer_id and seller_id are foreign keys to the Users table.

Table: Items

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| item_id       | int     |
| item_brand    | varchar |
+---------------+---------+
```

item_id is the primary key of this table.

Write an SQL query to find for each user, the join date and the number of orders they made as a buyer in **2019**.

The query result format is in the following example:

Users table:

```
+---------+------------+----------------+
| user_id | join_date  | favorite_brand |
+---------+------------+----------------+
| 1       | 2018-01-01 | Lenovo         |
| 2       | 2018-02-09 | Samsung        |
| 3       | 2018-01-19 | LG             |
| 4       | 2018-05-21 | HP             |
+---------+------------+----------------+
```

Orders table:

```
+----------+------------+---------+----------+-----------+
| order_id | order_date | item_id | buyer_id | seller_id |
+----------+------------+---------+----------+-----------+
| 1        | 2019-08-01 | 4       | 1        | 2         |
| 2        | 2018-08-02 | 2       | 1        | 3         |
| 3        | 2019-08-03 | 3       | 2        | 3         |
| 4        | 2018-08-04 | 1       | 4        | 2         |
| 5        | 2018-08-04 | 1       | 3        | 4         |
| 6        | 2019-08-05 | 2       | 2        | 4         |
+----------+------------+---------+----------+-----------+
```

Items table:

```
+---------+------------+
| item_id | item_brand |
+---------+------------+
| 1       | Samsung    |
| 2       | Lenovo     |
| 3       | LG         |
| 4       | HP         |
+---------+------------+
```

Result table:

```
+-----------+------------+----------------+
| buyer_id  | join_date  | orders_in_2019 |
+-----------+------------+----------------+
| 1         | 2018-01-01 | 1              |
| 2         | 2018-02-09 | 2              |
| 3         | 2018-01-19 | 0              |
| 4         | 2018-05-21 | 0              |
+-----------+------------+----------------+
```

```
SELECT U.user_id AS buyer_id, U.join_date, COUNT(item_id) AS orders_in_2019
FROM Users U LEFT JOIN

(SELECT order_date, item_id, buyer_id
 FROM Orders
 WHERE order_date BETWEEN '2019-01-01' AND '2019-12-31') O

ON U.user_id = O.buyer_id
GROUP BY U.user_id;
```

```
SELECT u.user_id AS buyer_id, join_date,

IFNULL(COUNT(order_date), 0) AS orders_in_2019

FROM Users as u LEFT JOIN Orders as o ON u.user_id = o.buyer_id

AND YEAR(order_date) = '2019'

GROUP BY u.user_id
```

1164. Product Price at a Given Date

Table: Products

```
+---------------+---------+

| Column Name   | Type    |

+---------------+---------+

| product_id    | int     |

| new_price     | int     |

| change_date   | date    |

+---------------+---------+
```

(product_id, change_date) is the primary key of this table.

Each row of this table indicates that the price of some product was changed to a new price at some date.

Write an SQL query to find the prices of all products on **2019-08-16**. Assume the price of all products before any change is **10**.

The query result format is in the following example:

Products table:

```
+------------+-----------+-------------+
| product_id | new_price | change_date |
+------------+-----------+-------------+
| 1          | 20        | 2019-08-14  |
| 2          | 50        | 2019-08-14  |
| 1          | 30        | 2019-08-15  |
| 1          | 35        | 2019-08-16  |
| 2          | 65        | 2019-08-17  |
| 3          | 20        | 2019-08-18  |
+------------+-----------+-------------+
```

Result table:

```
+------------+-------+
| product_id | price |
+------------+-------+
| 2          | 50    |
| 1          | 35    |
| 3          | 10    |
+------------+-------+
SELECT DISTINCT a.product_id,ifnull(temp.new_price,10) AS price
FROM products a LEFT JOIN
(SELECT *
FROM products
WHERE (product_id, change_date)
 IN
 (SELECT product_id, MAX(change_date)
  FROM products
  WHERE change_date <= "2019-08-16"
  GROUP BY product_id)) AS temp
ON a.product_id = temp.product_id;
```

1174. Immediate Food Delivery II

Table: Delivery

```
+-----------------------------+---------+
| Column Name                 | Type    |
+-----------------------------+---------+
| delivery_id                 | int     |
| customer_id                 | int     |
| order_date                  | date    |
| customer_pref_delivery_date | date    |
+-----------------------------+---------+
```

delivery_id is the primary key of this table.

The table holds information about food delivery to customers that make orders at some date and specify a preferred delivery date (on the same order date or after it).

If the preferred delivery date of the customer is the same as the order date then the order is called *immediate* otherwise it's called *scheduled*.

The *first order* of a customer is the order with the earliest order date that customer made. It is guaranteed that a customer has exactly one first order.

Write an SQL query to find the percentage of immediate orders in the first orders of all customers, **rounded to 2 decimal places**.

The query result format is in the following example:

Delivery table:

```
+-------------+-------------+------------+-----------------------------+
| delivery_id | customer_id | order_date | customer_pref_delivery_date |
+-------------+-------------+------------+-----------------------------+
| 1           | 1           | 2019-08-01 | 2019-08-02                  |
| 2           | 2           | 2019-08-02 | 2019-08-02                  |
| 3           | 1           | 2019-08-11 | 2019-08-12                  |
| 4           | 3           | 2019-08-24 | 2019-08-24                  |
| 5           | 3           | 2019-08-21 | 2019-08-22                 |
| 6           | 2           | 2019-08-11 | 2019-08-13                 |
| 7           | 4           | 2019-08-09 | 2019-08-09                 |
+-------------+-------------+------------+-----------------------------+
```

Result table:

```
+----------------------+
| immediate_percentage |
+----------------------+
| 50.00                |
+----------------------+
```

The customer id 1 has a first order with delivery id 1 and it is scheduled.

The customer id 2 has a first order with delivery id 2 and it is immediate.

The customer id 3 has a first order with delivery id 5 and it is scheduled.

The customer id 4 has a first order with delivery id 7 and it is immediate.

Hence, half the customers have immediate first orders.

```
SELECT ROUND(100*SUM(CASE WHEN order_date = customer_pref_delivery_date
            THEN 1
            ELSE 0
            END)/ COUNT(distinct customer_id),2) AS immediate_percentage

FROM Delivery
WHERE (customer_id, order_date)
IN
(SELECT customer_id, MIN(order_date) AS min_date
FROM Delivery
GROUP BY customer_id);
```

1193. Monthly Transactions I

Table: Transactions

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| country       | varchar |
| state         | enum    |
| amount        | int     |

| trans_date    | date    |

+---------------+---------+
```

id is the primary key of this table.

The table has information about incoming transactions.

The state column is an enum of type ["approved", "declined"].

Write an SQL query to find for each month and country, the number of transactions and their total amount, the number of approved transactions and their total amount.

The query result format is in the following example:

Transactions table:

```
+------+---------+----------+--------+------------+

| id   | country | state    | amount | trans_date |

+------+---------+----------+--------+------------+

| 121  | US      | approved | 1000   | 2018-12-18 |

| 122  | US      | declined | 2000   | 2018-12-19 |

| 123  | US      | approved | 2000   | 2019-01-01 |

| 124  | DE      | approved | 2000   | 2019-01-07 |

+------+---------+----------+--------+------------+
```

Result table:

```
+----------+---------+-------------+----------------+--------------------+---------------------
--+

| month    | country | trans_count | approved_count | trans_total_amount |
approved_total_amount |

+----------+---------+-------------+----------------+--------------------+---------------------
--+

| 2018-12  | US      | 2           | 1              | 3000               | 1000                  |

| 2019-01  | US      | 1           | 1              | 2000               | 2000                  |

| 2019-01  | DE      | 1           | 1              | 2000               | 2000                  |

+----------+---------+-------------+----------------+--------------------+---------------------
--+
```

```sql
SELECT LEFT(trans_date, 7) AS month, country, COUNT(id) AS
trans_count, SUM(state = 'approved') AS approved_count, SUM(amount)
AS trans_total_amount,

SUM(CASE

        WHEN state = 'approved' THEN amount

        ELSE 0

        END) AS approved_total_amount

FROM Transactions

GROUP BY month, country;
```

2021/1/19

1204. Last Person to Fit in the Elevator

Table: Queue

Column Name	Type
person_id	int
person_name	varchar
weight	int
turn	int

person_id is the primary key column for this table.

This table has the information about all people waiting for an elevator.

The person_id and turn columns will contain all numbers from 1 to n, where n is the number of rows in the table.

The maximum weight the elevator can hold is **1000**.

Write an SQL query to find the person_name of the last person who will fit in the elevator without exceeding the weight limit. It is guaranteed that the person who is first in the queue can fit in the elevator.

The query result format is in the following example:

Queue table

```
+-----------+-------------------+--------+------+
| person_id | person_name       | weight | turn |
+-----------+-------------------+--------+------+
| 5         | George Washington | 250    | 1    |
| 3         | John Adams        | 350    | 2    |
| 6         | Thomas Jefferson  | 400    | 3    |
| 2         | Will Johnliams    | 200    | 4    |
| 4         | Thomas Jefferson  | 175    | 5    |
| 1         | James Elephant    | 500    | 6    |
+-----------+-------------------+--------+------+
```

Result table

```
+-------------------+
| person_name       |
+-------------------+
| Thomas Jefferson  |
+-------------------+
```

Queue table is ordered by turn in the example for simplicity.

In the example George Washington(id 5), John Adams(id 3) and Thomas Jefferson(id 6) will enter the elevator as their weight sum is 250 + 350 + 400 = 1000.

Thomas Jefferson(id 6) is the last person to fit in the elevator because he has the last turn in these three people.

```
SELECT person_name
FROM Queue a
WHERE

(SELECT SUM(weight)
 FROM Queue b
 WHERE b.turn <= a.turn
 ORDER By turn) <= 1000

ORDER BY a.turn DESC LIMIT 1;
```

1205. Monthly Transactions II

Table: Transactions

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| id             | int     |
| country        | varchar |
| state          | enum    |
| amount         | int     |
| trans_date     | date    |
+----------------+---------+
```

id is the primary key of this table.

The table has information about incoming transactions.

The state column is an enum of type ["approved", "declined"].

Table: Chargebacks

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| trans_id       | int     |
| charge_date    | date    |
+----------------+---------+
```

Chargebacks contains basic information regarding incoming chargebacks from some transactions placed in Transactions table.

trans_id is a foreign key to the id column of Transactions table.

Each chargeback corresponds to a transaction made previously even if they were not approved.

Write an SQL query to find for each month and country, the number of approved transactions and their total amount, the number of chargebacks and their total amount.

Note: In your query, given the month and country, ignore rows with all zeros.

The query result format is in the following example:

Transactions table:

```
+------+---------+----------+--------+------------+
| id   | country | state    | amount | trans_date |
+------+---------+----------+--------+------------+
| 101  | US      | approved | 1000   | 2019-05-18 |
| 102  | US      | declined | 2000   | 2019-05-19 |
| 103  | US      | approved | 3000   | 2019-06-10 |
| 104  | US      | approved | 4000   | 2019-06-13 |
| 105  | US      | approved | 5000   | 2019-06-15 |
+------+---------+----------+--------+------------+
```

Chargebacks table:

```
+------------+------------+
| trans_id   | trans_date |
+------------+------------+
| 102        | 2019-05-29 |
| 101        | 2019-06-30 |
| 105        | 2019-09-18 |
+------------+------------+
```

Result table:

```
+----------+---------+----------------+-----------------+-------------------+------------------
--+
| month    | country | approved_count | approved_amount |
chargeback_count  | chargeback_amount  |
+----------+---------+----------------+-----------------+-------------------+------------------
--+
| 2019-05  | US      | 1              | 1000            | 1                 | 2000               |
| 2019-06  | US      | 3              | 12000           | 1                 | 1000               |
| 2019-09  | US      | 0              | 0               | 1                 | 5000               |
+----------+---------+----------------+-----------------+-------------------+-------
```

```
SELECT month, country, SUM(CASE WHEN state = "approved" THEN 1
ELSE 0 END) AS approved_count, SUM(CASE WHEN state = "approved"
THEN amount ELSE 0 END) AS approved_amount, SUM(CASE WHEN
state = "declined" THEN 1 ELSE 0 END) AS chargeback_count,
SUM(CASE WHEN state = "declined" THEN amount ELSE 0 END) AS
chargeback_amount

FROM


(SELECT LEFT(chargebacks.trans_date, 7) AS month, country, "declined"
AS state, amount

 FROM chargebacks JOIN transactions ON chargebacks.trans_id =
transactions.id

 UNION ALL

 SELECT LEFT(trans_date, 7) AS month, country, state, amount

 FROM transactions

 WHERE state = "approved") s


GROUP BY month, country;
```

1212. Team Scores in Football Tournament

Table: Teams

```
+---------------+----------+

| Column Name   | Type     |
+---------------+----------+
| team_id       | int      |
| team_name     | varchar  |
+---------------+----------+
```

team_id is the primary key of this table.

Each row of this table represents a single football team.

Table: Matches

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| match_id      | int     |
| host_team     | int     |
| guest_team    | int     |
| host_goals    | int     |
| guest_goals   | int     |
+---------------+---------+
```

match_id is the primary key of this table.

Each row is a record of a finished match between two different teams.

Teams host_team and guest_team are represented by their IDs in the teams table (team_id) and they scored host_goals and guest_goals goals respectively.

You would like to compute the scores of all teams after all matches. Points are awarded as follows:

- A team receives three points if they win a match (Score strictly more goals than the opponent team).
- A team receives one point if they draw a match (Same number of goals as the opponent team).
- A team receives no points if they lose a match (Score less goals than the opponent team).

Write an SQL query that selects the **team_id**, **team_name** and **num_points** of each team in the tournament after all described matches. Result table should be ordered by **num_points** (decreasing order). In case of a tie, order the records by **team_id** (increasing order).

The query result format is in the following example:

Teams table:

```
+-----------+--------------+
| team_id   | team_name    |
+-----------+--------------+
| 10        | Leetcode FC  |
| 20        | NewYork FC   |
| 30        | Atlanta FC   |
| 40        | Chicago FC   |
| 50        | Toronto FC   |
+-----------+--------------+
```

Matches table:

```
+------------+--------------+---------------+-------------+--------------+
| match_id   | host_team    | guest_team    | host_goals  | guest_goals  |
+------------+--------------+---------------+-------------+--------------+
| 1          | 10           | 20            | 3           | 0            |
| 2          | 30           | 10            | 2           | 2            |
| 3          | 10           | 50            | 5           | 1            |
| 4          | 20           | 30            | 1           | 0            |
| 5          | 50           | 30            | 1           | 0            |
+------------+--------------+---------------+-------------+--------------+
```

Result table:

```
+------------+--------------+---------------+
| team_id    | team_name    | num_points    |
+------------+--------------+---------------+
| 10         | Leetcode FC  | 7             |
| 20         | NewYork FC   | 3             |
| 50         | Toronto FC   | 3             |
| 30         | Atlanta FC   | 1             |
| 40         | Chicago FC   | 0             |
+------------+--------------+---------------+
```

```
SELECT team_id,team_name,

SUM(CASE WHEN team_id = host_team AND host_goals > guest_goals
THEN 3 ELSE 0 END)+

SUM(CASE WHEN team_id = guest_team AND guest_goals > host_goals
THEN 3 ELSE 0 END)+

SUM(CASE WHEN team_id = host_team AND host_goals = guest_goals
THEN 1 ELSE 0 END)+

SUM(CASE WHEN team_id = guest_team AND guest_goals = host_goals
THEN 1 ELSE 0 END)

AS num_points

FROM Teams

LEFT JOIN Matches ON team_id = host_team OR team_id = guest_team

GROUP BY team_id

ORDER BY num_points DESC, team_id ASC;
```

1264. Page Recommendations

Table: Friendship

```
+---------------+---------+

| Column Name   | Type    |

+---------------+---------+
| user1_id      | int     |

| user2_id      | int     |

+---------------+---------+
```

(user1_id, user2_id) is the primary key for this table.

Each row of this table indicates that there is a friendship relation between user1_id and user2_id.

Table: Likes

```
+-------------+---------+

| Column Name | Type    |

+-------------+---------+

| user_id     | int     |

| page_id     | int     |

+-------------+---------+
```

(user_id, page_id) is the primary key for this table.

Each row of this table indicates that user_id likes page_id.

Write an SQL query to recommend pages to the user with user_id = 1 using the pages that your friends liked. It should not recommend pages you already liked.

Return result table in any order without duplicates.

The query result format is in the following example:

Friendship table:

```
+----------+----------+
| user1_id | user2_id |
+----------+----------+
| 1        | 2        |
| 1        | 3        |
| 1        | 4        |
| 2        | 3        |
| 2        | 4        |
| 2        | 5        |
| 6        | 1        |
+----------+----------+
```

Likes table:

```
+---------+---------+
| user_id | page_id |
+---------+---------+
| 1       | 88      |
| 2       | 23      |
| 3       | 24      |
| 4       | 56      |
| 5       | 11      |
| 6       | 33      |
| 2       | 77      |
| 3       | 77      |
| 6       | 88      |
+---------+---------+
```

Result table:

```
+------------------+
| recommended_page |
+------------------+
| 23               |
| 24               |
| 56               |
| 33               |
| 77               |
+------------------+
```

User one is friend with users 2, 3, 4 and 6.

Suggested pages are 23 from user 2, 24 from user 3, 56 from user 3 and 33 from user 6.

Page 77 is suggested from both user 2 and user 3.

Page 88 is not suggested because user 1 already likes it.

```
SELECT DISTINCT page_id AS recommended_page

FROM


(SELECT CASE WHEN user1_id = 1 THEN user2_id

        WHEN user2_id = 1 THEN user1_id

        END AS user_id

 FROM Friendship) a JOIN Likes ON a.user_id = Likes.user_id


WHERE page_id NOT IN (SELECT page_id FROM Likes WHERE user_id
= 1);
```

1270. All People Report to the Given Manager

Table: Employees

```
+---------------+---------+

| Column Name   | Type    |

+---------------+---------+

| employee_id   | int     |

| employee_name | varchar |
| manager_id    | int     |
+---------------+---------+
```

employee_id is the primary key for this table.

Each row of this table indicates that the employee with ID employee_id and name employee_name reports his work to his/her direct manager with manager_id

The head of the company is the employee with employee_id = 1.

Write an SQL query to find employee_id of all employees that directly or indirectly report their work to the head of the company.

The indirect relation between managers will not exceed 3 managers as the company is small.

Return result table in any order without duplicates.

The query result format is in the following example:

Employees table:

```
+-------------+---------------+------------+
| employee_id | employee_name | manager_id |
+-------------+---------------+------------+
| 1           | Boss          | 1          |
| 3           | Alice         | 3          |
| 2           | Bob           | 1          |
| 4           | Daniel        | 2          |
| 7           | Luis          | 4          |
| 8           | Jhon          | 3          |
| 9           | Angela        | 8          |
| 77          | Robert        | 1          |
+-------------+---------------+------------+
```

Result table:

```
+-------------+
| employee_id |
+-------------+
| 2           |
| 77          |
| 4           |
| 7           |
+-------------+
```

The head of the company is the employee with employee_id 1.

The employees with employee_id 2 and 77 report their work directly to the head of the company.

The employee with employee_id 4 report his work indirectly to the head of the company 4 --> 2 --> 1.

The employee with employee_id 7 report his work indirectly to the head of the company 7 --> 4 --> 2 --> 1.

The employees with employee_id 3, 8 and 9 don't report their work to head of company directly or indirectly.

```
SELECT e1.employee_id
FROM Employees e1 JOIN Employees e2 ON e1.manager_id = e2.employee_id
                  JOIN Employees e3 ON e2.manager_id = e3.employee_id
WHERE e3.manager_id = 1 AND e1.employee_id != 1;
```

1285. Find the Start and End Number of Continuous Ranges

Table: Logs

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| log_id        | int     |
+---------------+---------+
```

id is the primary key for this table.

Each row of this table contains the ID in a log Table.

Since some IDs have been removed from Logs. Write an SQL query to find the start and end number of continuous ranges in table Logs.

Order the result table by start_id.

The query result format is in the following example:

Logs table:

```
+------------+
| log_id     |
+------------+
| 1          |
| 2          |
| 3          |
| 7          |
| 8          |
| 10         |
+------------+
```

Result table:

```
+------------+--------------+
| start_id   | end_id       |
+------------+--------------+
| 1          | 3            |
| 7          | 8            |
| 10         | 10           |
+------------+--------------+
```

The result table should contain all ranges in table Logs.

From 1 to 3 is contained in the table.

From 4 to 6 is missing in the table

From 7 to 8 is contained in the table.

Number 9 is missing in the table.

Number 10 is contained in the table.

```
SELECT MIN(log_id) AS start_id, MAX(log_id) AS end_id
FROM

(SELECT log_id, ROW_NUMBER() OVER(ORDER BY log_id) AS num
 FROM Logs) a
 GROUP BY (log_id - num)
 ORDER BY start_id;
```

1308. Running Total for Different Genders

Table: Scores

```
+---------------+---------+
| Column Name   | Type    |

+---------------+---------+

| player_name   | varchar |

| gender        | varchar |

| day           | date    |

| score_points  | int     |

+---------------+---------+
```

(gender, day) is the primary key for this table.

A competition is held between females team and males team.

Each row of this table indicates that a player_name and with gender has scored score_point in someday.

Gender is 'F' if the player is in females team and 'M' if the player is in males team.

Write an SQL query to find the total score for each gender at each day.

Order the result table by gender and day

The query result format is in the following example:

Scores table:

```
+-------------+--------+------------+--------------+

| player_name | gender | day        | score_points |

+-------------+--------+------------+--------------+
| Aron        | F      | 2020-01-01 | 17           |
| Alice       | F      | 2020-01-07 | 23           |
| Bajrang     | M      | 2020-01-07 | 7            |
| Khali       | M      | 2019-12-25 | 11           |
| Slaman      | M      | 2019-12-30 | 13           |
| Joe         | M      | 2019-12-31 | 3            |
| Jose        | M      | 2019-12-18 | 2            |
| Priya       | F      | 2019-12-31 | 23           |
| Priyanka    | F      | 2019-12-30 | 17           |
+-------------+--------+------------+--------------+
```

Result table:

```
+--------+------------+-------+
| gender | day        | total |
+--------+------------+-------+
| F      | 2019-12-30 | 17    |
| F      | 2019-12-31 | 40    |
| F      | 2020-01-01 | 57    |
| F      | 2020-01-07 | 80    |
| M      | 2019-12-18 | 2     |
| M      | 2019-12-25 | 13    |
| M      | 2019-12-30 | 26    |
| M      | 2019-12-31 | 29    |
| M      | 2020-01-07 | 36    |
+--------+------------+-------+
```

For females team:

First day is 2019-12-30, Priyanka scored 17 points and the total score for the team is 17.

Second day is 2019-12-31, Priya scored 23 points and the total score for the team is 40.

Third day is 2020-01-01, Aron scored 17 points and the total score for the team is 57.

Fourth day is 2020-01-07, Alice scored 23 points and the total score for the team is 80.

For males team:

First day is 2019-12-18, Jose scored 2 points and the total score for the team is 2.

Second day is 2019-12-25, Khali scored 11 points and the total score for the team is 13.

Third day is 2019-12-30, Slaman scored 13 points and the total score for the team is 26.

Fourth day is 2019-12-31, Joe scored 3 points and the total score for the team is 29.

Fifth day is 2020-01-07, Bajrang scored 7 points and the total score for the team is 36.

```
SELECT gender, day, SUM(score_points) OVER(PARTITION BY gender ORDER BY day) AS total

FROM Scores;
```

1321. Restaurant Growth

Table: Customer

```
+---------------+---------+

| Column Name   | Type    |

+---------------+---------+

| customer_id   | int     |

| name          | varchar |

| visited_on    | date    |

| amount        | int     |

+---------------+---------+
```

(customer_id, visited_on) is the primary key for this table.

This table contains data about customer transactions in a restaurant.

visited_on is the date on which the customer with ID (customer_id) have visited the restaurant.

amount is the total paid by a customer.

You are the restaurant owner and you want to analyze a possible expansion (there will be at least one customer every day).

Write an SQL query to compute moving average of how much customer paid in a 7 days window (current day + 6 days before) .

The query result format is in the following example:

Return result table ordered by visited_on.

average_amount should be **rounded to 2 decimal places**, all dates are in the format ('YYYY-MM-DD').

Customer table:

```
+-------------+--------------+--------------+-------------+

| customer_id | name         | visited_on   | amount      |

+-------------+--------------+--------------+-------------+

| 1           | Jhon         | 2019-01-01   | 100         |

| 2           | Daniel       | 2019-01-02   | 110         |

| 3           | Jade         | 2019-01-03   | 120         |

| 4           | Khaled       | 2019-01-04   | 130         |

| 5           | Winston      | 2019-01-05   | 110         |

| 6           | Elvis        | 2019-01-06   | 140         |

| 7           | Anna         | 2019-01-07   | 150         |

| 8           | Maria        | 2019-01-08   | 80          |

| 9           | Jaze         | 2019-01-09   | 110         |
| 1           | Jhon         | 2019-01-10   | 130         |

| 3           | Jade         | 2019-01-10   | 150         |

+-------------+--------------+--------------+-------------+
```

Result table:

```
+--------------+--------------+----------------+

| visited_on   | amount       | average_amount |

+--------------+--------------+----------------+

| 2019-01-07   | 860          | 122.86         |

| 2019-01-08   | 840          | 120            |

| 2019-01-09   | 840          | 120            |

| 2019-01-10   | 1000         | 142.86         |

+--------------+--------------+----------------+
```

1st moving average from 2019-01-01 to 2019-01-07 has an average_amount of (100 + 110 + 120 + 130 + 110 + 140 + 150)/7 = 122.86

2nd moving average from 2019-01-02 to 2019-01-08 has an average_amount of (110 + 120 + 130 + 110 + 140 + 150 + 80)/7 = 120

3rd moving average from 2019-01-03 to 2019-01-09 has an average_amount of (120 + 130 + 110 + 140 + 150 + 80 + 110)/7 = 120

4th moving average from 2019-01-04 to 2019-01-10 has an average_amount of (130 + 110 + 140 + 150 + 80 + 110 + 130 + 150)/7 = 142.86

```
SELECT a.visited_on AS visited_on, SUM(b.day_sum) AS amount,

        ROUND(AVG(b.day_sum), 2) AS average_amount

FROM

  (SELECT visited_on, SUM(amount) AS day_sum FROM Customer GROUP BY visited_on ) a,

  (SELECT visited_on, SUM(amount) AS day_sum FROM Customer GROUP BY visited_on ) b

WHERE DATEDIFF(a.visited_on, b.visited_on) BETWEEN 0 AND 6

GROUP BY a.visited_on

HAVING COUNT(b.visited_on) = 7;
```

1341. Movie Rating

Table: Movies

```
+---------------+---------+

| Column Name   | Type    |

+---------------+---------+

| movie_id      | int     |

| title         | varchar |

+---------------+---------+
```

movie_id is the primary key for this table.

title is the name of the movie.

Table: Users

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| name          | varchar |
+---------------+---------+
```

user_id is the primary key for this table.

Table: Movie_Rating

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| movie_id      | int     |
| user_id       | int     |
| rating        | int     |
| created_at    | date    |
+---------------+---------+
```

(movie_id, user_id) is the primary key for this table.

This table contains the rating of a movie by a user in their review.

created_at is the user's review date.

Write the following SQL query:

- Find the name of the user who has rated the greatest number of movies.
  In case of a tie, return lexicographically smaller user name.

- Find the movie name with the ***highest average*** rating in **February 2020**.
  In case of a tie, return lexicographically smaller movie name.

The query is returned in 2 rows, the query result format is in the following example:

Movies table:

```
+-------------+--------------+
| movie_id    |  title       |
+-------------+--------------+
| 1           | Avengers     |
| 2           | Frozen 2     |
| 3           | Joker        |
+-------------+--------------+
```

Users table:

```
+-------------+--------------+
| user_id     |  name        |
+-------------+--------------+
| 1           | Daniel       |
| 2           | Monica       |
| 3           | Maria        |
| 4           | James        |
+-------------+--------------+
```

Movie_Rating table:

```
+-------------+--------------+--------------+-------------+
| movie_id    | user_id      | rating       | created_at  |
+-------------+--------------+--------------+-------------+
| 1           | 1            | 3            | 2020-01-12  |
| 1           | 2            | 4            | 2020-02-11  |
| 1           | 3            | 2            | 2020-02-12  |
| 1           | 4            | 1            | 2020-01-01  |
| 2           | 1            | 5            | 2020-02-17  |
| 2           | 2            | 2            | 2020-02-01  |
| 2           | 3            | 2            | 2020-03-01  |
| 3           | 1            | 3            | 2020-02-22  |
| 3           | 2            | 4            | 2020-02-25  |
+-------------+--------------+--------------+-------------+
```

Result table:

```
+--------------+
| results      |
+--------------+
| Daniel       |
| Frozen 2     |
+--------------+
```

Daniel and Monica have rated 3 movies ("Avengers", "Frozen 2" and "Joker") but Daniel is smaller lexicographically.

Frozen 2 and Joker have a rating average of 3.5 in February but Frozen 2 is smaller lexicographically.

```
(SELECT name AS results
 FROM users u JOIN movie_rating m ON u.user_id = m.user_id
 GROUP BY results
 ORDER BY count(rating) desc, results ASC LIMIT 1)
UNION
```

```
(SELECT title AS results
 FROM movies m1 JOIN movie_rating m2 ON m1.movie_id = m2.movie_id
 WHERE month(created_at) = 2
 GROUP BY results
 ORDER BY AVG(rating) DESC, results ASC LIMIT 1);
```

1355. Activity Participants

Table: Friends

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| name          | varchar |
| activity      | varchar |
+---------------+---------+
```

id is the id of the friend and primary key for this table.

name is the name of the friend.

activity is the name of the activity which the friend takes part in.

Table: Activities

```
+---------------+---------+

| Column Name   | Type    |

+---------------+---------+

| id            | int     |

| name          | varchar |

+---------------+---------+
```

id is the primary key for this table.

name is the name of the activity.

Write an SQL query to find the names of all the activities with neither maximum, nor minimum number of participants.

Return the result table in any order. Each activity in table Activities is performed by any person in the table Friends.

The query result format is in the following example:

Friends table:

```
+------+--------------+---------------+

| id   | name         | activity      |

+------+--------------+---------------+

| 1    | Jonathan D.  | Eating        |

| 2    | Jade W.      | Singing       |

| 3    | Victor J.    | Singing       |
| 4    | Elvis Q.     | Eating        |
| 5    | Daniel A.    | Eating        |
| 6    | Bob B.       | Horse Riding  |
+------+--------------+---------------+
```

Activities table:

```
+------+--------------+
| id   | name         |
+------+--------------+
| 1    | Eating       |
| 2    | Singing      |
| 3    | Horse Riding |
+------+--------------+
```

Result table:

```
+--------------+
| activity     |
+--------------+
| Singing      |
+--------------+
```

Eating activity is performed by 3 friends, maximum number of participants, (Jonathan D. , Elvis Q. and Daniel A.)

Horse Riding activity is performed by 1 friend, minimum number of participants, (Bob B.)

Singing is performed by 2 friends (Victor J. and Jade W.)

```
SELECT activity
FROM friends
GROUP BY activity

HAVING COUNT(*) >
(SELECT COUNT(*) AS NUM
 FROM friends
 GROUP BY activity
 ORDER BY NUM ASC LIMIT 1)

AND COUNT(*) <
(SELECT COUNT(*) AS NUM
 FROM friends
 GROUP BY activity
 ORDER BY NUM DESC LIMIT 1)
```

2021/1/20

1364. Number of Trusted Contacts of a Customer

Table: Customers

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| customer_id   | int     |
| customer_name | varchar |
| email         | varchar |
+---------------+---------+
```

customer_id is the primary key for this table.

Each row of this table contains the name and the email of a customer of an online shop.

Table: Contacts

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | id      |
| contact_name  | varchar |
| contact_email | varchar |
+---------------+---------+
```

(user_id, contact_email) is the primary key for this table.

Each row of this table contains the name and email of one contact of customer with user_id.

This table contains information about people each customer trust. The contact may or may not exist in the Customers table.

Table: Invoices

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| invoice_id   | int     |
| price        | int     |
| user_id      | int     |
+--------------+---------+
```

invoice_id is the primary key for this table.

Each row of this table indicates that user_id has an invoice with invoice_id and a price.

Write an SQL query to find the following for each invoice_id:

- customer_name: The name of the customer the invoice is related to.
- price: The price of the invoice.
- contacts_cnt: The number of contacts related to the customer.
- trusted_contacts_cnt: The number of contacts related to the customer and at the same time they are customers to the shop. (i.e His/Her email exists in the Customers table.)

Order the result table by `invoice_id`.

The query result format is in the following example:

Customers table:

```
+-------------+---------------+--------------------+

| customer_id | customer_name | email              |

+-------------+---------------+--------------------+

| 1           | Alice         | alice@leetcode.com |

| 2           | Bob           | bob@leetcode.com   |

| 13          | John          | john@leetcode.com  |

| 6           | Alex          | alex@leetcode.com  |

+-------------+---------------+--------------------+
```

Contacts table:

```
+-------------+--------------+--------------------+

| user_id     | contact_name | contact_email      |

+-------------+--------------+--------------------+

| 1           | Bob          | bob@leetcode.com   |
| 1           | John         | john@leetcode.com  |
| 1           | Jal          | jal@leetcode.com   |
| 2           | Omar         | omar@leetcode.com  |
| 2           | Meir         | meir@leetcode.com  |
| 6           | Alice        | alice@leetcode.com |
+-------------+--------------+--------------------+
```

Invoices table:

```
+------------+-------+---------+
| invoice_id | price | user_id |
+------------+-------+---------+
| 77         | 100   | 1       |
| 88         | 200   | 1       |
| 99         | 300   | 2       |
| 66         | 400   | 2       |
| 55         | 500   | 13      |
| 44         | 60    | 6       |
+------------+-------+---------+
```

Result table:

```
+------------+---------------+-------+--------------+----------------------+
| invoice_id | customer_name | price | contacts_cnt | trusted_contacts_cnt |
+------------+---------------+-------+--------------+----------------------+
| 44         | Alex          | 60    | 1            | 1                    |
| 55         | John          | 500   | 0            | 0                    |
| 66         | Bob           | 400   | 2            | 0                    |
| 77         | Alice         | 100   | 3            | 2                    |
| 88         | Alice         | 200   | 3            | 2                    |
| 99         | Bob           | 300   | 2            | 0                    |
+------------+---------------+-------+--------------+----------------------+
```

Alice has three contacts, two of them are trusted contacts (Bob and John).

Bob has two contacts, none of them is a trusted contact.

Alex has one contact and it is a trusted contact (Alice).

John doesn't have any contacts.

```
SELECT i.invoice_id, c.customer_name, i.price, COUNT(con.user_id) AS
contacts_cnt, COUNT(c2.email) AS trusted_contacts_cnt

FROM invoices i LEFT JOIN customers c ON c.customer_id = i.user_id

                LEFT JOIN contacts con ON con.user_id = c.customer_id

                LEFT JOIN customers c2 ON c2.email = con.contact_email

GROUP BY i.invoice_id

ORDER BY i.invoice_id;
```

1393. Capital Gain/Loss

Table: Stocks

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| stock_name    | varchar |
| operation     | enum    |
| operation_day | int     |
| price         | int     |
+---------------+---------+
```

(stock_name, operation_day) is the primary key for this table.

The operation column is an ENUM of type ('Sell', 'Buy')

Each row of this table indicates that the stock which has stock_name had an operation on the day operation_day with the price.

It is guaranteed that each 'Sell' operation for a stock has a corresponding 'Buy' operation in a previous day.

Write an SQL query to report the Capital gain/loss for each stock.

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

Return the result table in any order.

The query result format is in the following example:

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  |
+---------------+-----------+---------------+--------+
```

Result table:

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

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$. It was bought again at day 4 for 1000$ and was sold at day 5 for 500$. At last, it was bought at day 6 for 1000$ and was sold at day 10 for 10000$. Capital gain/loss is the sum of capital gains/losses for each ('Buy' --> 'Sell') operation = (1010 - 10) + (500 - 1000) + (10000 - 1000) = 1000 - 500 + 9000 = 9500$.

```
SELECT stock_name, SUM(CASE WHEN operation = 'Buy' THEN -price
                            ELSE price END) AS capital_gain_loss
FROM Stocks
GROUP BY stock_name;
```

1398. Customers Who Bought Products A and B but Not C

Table: Customers

```
+---------------------+---------+
| Column Name         | Type    |
+---------------------+---------+
| customer_id         | int     |
| customer_name       | varchar |
+---------------------+---------+
```

customer_id is the primary key for this table.

customer_name is the name of the customer.

Table: Orders

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| order_id      | int     |
| customer_id   | int     |
| product_name  | varchar |
+---------------+---------+
```

order_id is the primary key for this table.

customer_id is the id of the customer who bought the product "product_name".

Write an SQL query 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 buy this product.

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

The query result format is in the following example.

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         |
+------------+--------------+---------------+
```

Result table:

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

Only the customer_id with id 3 bought the product A and B but not the product C.

```
SELECT a.customer_id, a.customer_name
FROM customers a JOIN orders b ON a.customer_id  = b.customer_id
GROUP BY a.customer_id
HAVING SUM(b.product_name="A") >0 AND SUM(b.product_name="B") > 0 AND SUM(b.product_name="C")=0;
```

1421. NPV Queries

Table: NPV

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| year          | int     |
| npv           | int     |
+---------------+---------+
```

(id, year) is the primary key of this table.

The table has information about the id and the year of each inventory and the corresponding net present value.

Table: Queries

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| year          | int     |
+---------------+---------+
```

(id, year) is the primary key of this table.

The table has information about the id and the year of each inventory query.

Write an SQL query to find the npv of all each query of queries table.

Return the result table in any order.

The query result format is in the following example:

NPV table:

```
+------+--------+--------+
| id   | year   | npv    |
+------+--------+--------+
| 1    | 2018   | 100    |
| 7    | 2020   | 30     |
| 13   | 2019   | 40     |
| 1    | 2019   | 113    |
| 2    | 2008   | 121    |
| 3    | 2009   | 12     |
| 11   | 2020   | 99     |
| 7    | 2019   | 0      |
+------+--------+--------+
```

Queries table:

```
+------+--------+
| id   | year   |
+------+--------+
| 1    | 2019   |
| 2    | 2008   |
| 3    | 2009   |
| 7    | 2018   |
| 7    | 2019   |
| 7    | 2020   |
| 13   | 2019   |
+------+--------+
```

Result table:

```
+------+--------+--------+
| id   | year   | npv    |
+------+--------+--------+
| 1    | 2019   | 113    |
| 2    | 2008   | 121    |
| 3    | 2009   | 12     |
| 7    | 2018   | 0      |
| 7    | 2019   | 0      |
| 7    | 2020   | 30     |
| 13   | 2019   | 40     |
+------+--------+--------+
```

The npv value of (7, 2018) is not present in the NPV table, we consider it 0.

The npv values of all other queries can be found in the NPV table.

```
SELECT t1.id, t1.year, IFNULL(npv, 0) AS npv
FROM queries t1 LEFT JOIN NPV t2
ON t1.id = t2.id AND t1.year = t2.year;
```

1440. Evaluate Boolean Expression

Table Variables:

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| name          | varchar |
| value         | int     |
+---------------+---------+
```

name is the primary key for this table.

This table contains the stored variables and their values.

Table Expressions:

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| left_operand  | varchar |
| operator      | enum    |
| right_operand | varchar |
+---------------+---------+
```

(left_operand, operator, right_operand) is the primary key for this table.

This table contains a boolean expression that should be evaluated.

operator is an enum that takes one of the values ('<', '>', '=')

The values of left_operand and right_operand are guaranteed to be in the Variables table.

Write an SQL query to evaluate the boolean expressions in Expressions table.

Return the result table in any order.

The query result format is in the following example.

Variables table:

```
+------+-------+
| name | value |
+------+-------+
| x    | 66    |
| y    | 77    |
+------+-------+
```

Expressions table:

```
+--------------+----------+---------------+
| left_operand | operator | right_operand |
+--------------+----------+---------------+
| x            | >        | y             |
| x            | <        | y             |
| x            | =        | y             |
| y            | >        | x             |
| y            | <        | x             |
| x            | =        | x             |
+--------------+----------+---------------+
```

Result table:

```
+--------------+----------+---------------+-------+
| left_operand | operator | right_operand | value |
+--------------+----------+---------------+-------+
| x            | >        | y             | false |
| x            | <        | y             | true  |
| x            | =        | y             | false |
| y            | >        | x             | true  |
| y            | <        | x             | false |
| x            | =        | x             | true  |
+--------------+----------+---------------+-------+
```

As shown, you need find the value of each boolean exprssion in the table using the variables table.

```
SELECT e.left_operand, e.operator, e.right_operand,
(CASE WHEN e.operator = '<' AND v1.value < v2.value THEN 'true'
      WHEN e.operator = '=' AND v1.value = v2.value THEN 'true'
      WHEN e.operator = '>' AND v1.value > v2.value THEN 'true'
      ELSE 'false' END) AS value
FROM Expressions e LEFT JOIN Variables v1 ON e.left_operand = v1.name
          LEFT JOIN Variables v2 ON e.right_operand = v2.name;
```

1445. Apples & Oranges

Table: Sales

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| sale_date     | date    |
| fruit         | enum    |
| sold_num      | int     |
+---------------+---------+
```

(sale_date,fruit) is the primary key for this table.

This table contains the sales of "apples" and "oranges" sold each day.

Write an SQL query to report the difference between number of **apples** and **oranges** sold each day.

Return the result table **ordered** by sale_date in format ('YYYY-MM-DD').

The query result format is in the following example:

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          |
+------------+------------+-------------+
```

Result table:

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

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 - 0 = 20).

Day 2020-05-04, 15 apples and 16 oranges were sold (Difference 15 - 16 = -1).

```
SELECT sale_date, SUM(CASE WHEN fruit = "apples" THEN sold_num ELSE -sold_num END) AS diff

FROM sales

GROUP BY sale_date;
```

1454. Active Users

Table Accounts:

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| name          | varchar |
+---------------+---------+
```

the id is the primary key for this table.

This table contains the account id and the user name of each account.

Table Logins:

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| login_date    | date    |
+---------------+---------+
```

There is no primary key for this table, it may contain duplicates.

This table contains the account id of the user who logged in and the login date. A user may log in multiple times in the day.

Write an SQL query to find the id and the name of active users.

Active users are those who logged in to their accounts for 5 or more consecutive days.

Return the result table **ordered** by the id.

The query result format is in the following example:

Accounts table:

```
+----+----------+
| id | name     |
+----+----------+
| 1  | Winston  |
| 7  | Jonathan |
+----+----------+
```

Logins table:

```
+----+------------+
| id | login_date |
+----+------------+
| 7  | 2020-05-30 |
| 1  | 2020-05-30 |
| 7  | 2020-05-31 |
| 7  | 2020-06-01 |
| 7  | 2020-06-02 |
| 7  | 2020-06-02 |
| 7  | 2020-06-03 |
| 1  | 2020-06-07 |
| 7  | 2020-06-10 |
+----+------------+
```

Result table:

```
+----+----------+
| id | name     |
+----+----------+
| 7  | Jonathan |
+----+----------+
```

User Winston with id = 1 logged in 2 times only in 2 different days, so, Winston is not an active user.

User Jonathan with id = 7 logged in 7 times in 6 different days, five of them were consecutive days, so, Jonathan is an active user.

```
SELECT DISTINCT a.id, (SELECT name FROM Accounts WHERE id = a.id) AS name

FROM Logins a LEFT JOIN Logins b ON a.id = b.id

WHERE DATEDIFF(a.login_date,b.login_date) BETWEEN 1 AND 4

GROUP BY a.id, a.login_date

HAVING COUNT(DISTINCT b.login_date) = 4;
```

1459. Rectangles Area

Table: Points

```
+---------------+---------+

| Column Name   | Type    |

+---------------+---------+

| id            | int     |

| x_value       | int     |

| y_value       | int     |

+---------------+---------+
```

id is the primary key for this table.

Each point is represented as a 2D Dimensional (x_value, y_value).

Write an SQL query to report of all possible rectangles which can be formed by any two points of the table.

Each row in the result contains three columns (p1, p2, area) where:

- **p1** and **p2** are the id of two opposite corners of a rectangle and p1 < p2.
- Area of this rectangle is represented by the column **area**.

Report the query in descending order by area in case of tie in ascending order by p1 and p2.

Points table:

```
+----------+-------------+-------------+

| id       | x_value     | y_value     |

+----------+-------------+-------------+

| 1        | 2           | 8           |

| 2        | 4           | 7           |

| 3        | 2           | 10          |

+----------+-------------+-------------+
```

Result table:

```
+----------+-------------+-------------+

| p1       | p2          | area        |

+----------+-------------+-------------+

| 2        | 3           | 6           |

| 1        | 2           | 2           |

+----------+-------------+-------------+
```

p1 should be less than p2 and area greater than 0.

p1 = 1 and p2 = 2, has an area equal to |2-4| * |8-7| = 2.

p1 = 2 and p2 = 3, has an area equal to |4-2| * |7-10| = 6.

p1 = 1 and p2 = 3 It's not possible because the rectangle has an area equal to 0.

```
SELECT  a.id as P1, b.id as P2,

        ABS(b.x_value - a.x_value) * ABS(b.y_value - a.y_value) AS AREA

FROM Points a JOIN Points b ON a.id < b.id

WHERE a.x_value != b.x_value AND a.y_value != b.y_value

ORDER BY AREA DESC, p1 ASC, p2 ASC;
```

1468. Calculate Salaries

Table Salaries:

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| company_id    | int     |
| employee_id   | int     |
| employee_name | varchar |
| salary        | int     |
+---------------+---------+
```

(company_id, employee_id) is the primary key for this table.

This table contains the company id, the id, the name and the salary for an employee.

Write an SQL query to find the salaries of the employees after applying taxes.

The tax rate is calculated for each company based on the following criteria:

- 0% If the max salary of any employee in the company is less than 1000$.
- 24% If the max salary of any employee in the company is in the range [1000, 10000] inclusive.
- 49% If the max salary of any employee in the company is greater than 10000$.

Return the result table **in any order**. Round the salary to the nearest integer.

The query result format is in the following example:

Salaries table:

```
+------------+-------------+---------------+--------+

| company_id | employee_id | employee_name | salary |

+------------+-------------+---------------+--------+

| 1          | 1           | Tony          | 2000   |

| 1          | 2           | Pronub        | 21300  |
| 1          | 3           | Tyrrox        | 10800  |
| 2          | 1           | Pam           | 300    |
| 2          | 7           | Bassem        | 450    |
| 2          | 9           | Hermione      | 700    |
| 3          | 7           | Bocaben       | 100    |
| 3          | 2           | Ognjen        | 2200   |
| 3          | 13          | Nyancat       | 3300   |
| 3          | 15          | Morninngcat   | 7777   |
+------------+-------------+---------------+--------+
```

Result table:

```
+------------+-------------+---------------+--------+
| company_id | employee_id | employee_name | salary |
+------------+-------------+---------------+--------+
| 1          | 1           | Tony          | 1020   |
| 1          | 2           | Pronub        | 10863  |
| 1          | 3           | Tyrrox        | 5508   |
| 2          | 1           | Pam           | 300    |
| 2          | 7           | Bassem        | 450    |
| 2          | 9           | Hermione      | 700    |
| 3          | 7           | Bocaben       | 76     |
| 3          | 2           | Ognjen        | 1672   |
| 3          | 13          | Nyancat       | 2508   |
| 3          | 15          | Morninngcat   | 5911   |
+------------+-------------+---------------+--------+
```

For company 1, Max salary is 21300. Employees in company 1 have taxes = 49%

For company 2, Max salary is 700. Employees in company 2 have taxes = 0%

For company 3, Max salary is 7777. Employees in company 3 have taxes = 24%

The salary after taxes = salary - (taxes percentage / 100) * salary

For example, Salary for Morninngcat (3, 15) after taxes = 7777 - 7777 * (24 / 100) = 7777 - 1866.48 = 5910.52, which is rounded to 5911.

```
SELECT company_id, employee_id, employee_name,
ROUND(CASE WHEN MAX(salary) over(PARTITION BY company_id) < 1000 THEN salary
        WHEN MAX(salary) over(PARTITION BY company_id) BETWEEN 1000 AND 10000 THEN (1-0.24)*salary
        ELSE (1-0.49)*salary END,0)  AS salary
FROM salaries;
```

1501. Countries You Can Safely Invest In

Table Person:

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| id             | int     |
| name           | varchar |
| phone_number   | varchar |
+----------------+---------+
```

id is the primary key for this table.

Each row of this table contains the name of a person and their phone number.

Phone number will be in the form 'xxx-yyyyyyy' where xxx is the country code (3 characters) and yyyyyyy is the phone number (7 characters) where x and y are digits. Both can contain leading zeros.

Table Country:

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| name           | varchar |
| country_code   | varchar |
+----------------+---------+
```

country_code is the primary key for this table.

Each row of this table contains the country name and its code. country_code will be in the form 'xxx' where x is digits.

Table Calls:

```
+-------------+------+

| Column Name | Type |

+-------------+------+

| caller_id   | int  |

| callee_id   | int  |

| duration    | int  |

+-------------+------+
```

There is no primary key for this table, it may contain duplicates.

Each row of this table contains the caller id, callee id and the duration of the call in minutes. caller_id != callee_id

A telecommunications company wants to invest in new countries. The company intends to invest in the countries where the average call duration of the calls in this country is strictly greater than the global average call duration.

Write an SQL query to find the countries where this company can invest.

Return the result table in any order.

The query result format is in the following example.

Person table:

```
+----+----------+--------------+
| id | name     | phone_number |
+----+----------+--------------+
| 3  | Jonathan | 051-1234567  |
| 12 | Elvis    | 051-7654321  |
| 1  | Moncef   | 212-1234567  |
| 2  | Maroua   | 212-6523651  |
| 7  | Meir     | 972-1234567  |
| 9  | Rachel   | 972-0011100  |
+----+----------+--------------+
```

Country table:

```
+----------+--------------+
| name     | country_code |
+----------+--------------+
| Peru     | 051          |
| Israel   | 972          |
| Morocco  | 212          |
| Germany  | 049          |
| Ethiopia | 251          |
+----------+--------------+
```

Calls table:

```
+-----------+-----------+----------+
| caller_id | callee_id | duration |
+-----------+-----------+----------+
| 1         | 9         | 33       |
| 2         | 9         | 4        |
| 1         | 2         | 59       |
| 3         | 12        | 102      |
| 3         | 12        | 330      |
| 12        | 3         | 5        |
| 7         | 9         | 13       |
| 7         | 1         | 3        |
| 9         | 7         | 1        |
| 1         | 7         | 7        |
+-----------+-----------+----------+
```

Result table:

```
+----------+
| country  |
+----------+
| Peru     |
+----------+
```

The average call duration for Peru is (102 + 102 + 330 + 330 + 5 + 5) / 6 = 145.666667

The average call duration for Israel is (33 + 4 + 13 + 13 + 3 + 1 + 1 + 7) / 8 = 9.37500

The average call duration for Morocco is (33 + 4 + 59 + 59 + 3 + 7) / 6 = 27.5000

Global call duration average = (2 * (33 + 3 + 59 + 102 + 330 + 5 + 13 + 3 + 1 + 7)) / 20 = 55.70000

Since Peru is the only country where average call duration is greater than the global average, it's the only recommended country.

```
SELECT Country.name AS country
FROM Person JOIN Calls ON Calls.caller_id = Person.id OR Calls.callee_id = Person.id
        JOIN Country ON Country.country_code = LEFT(Person.phone_number, 3)
GROUP BY Country.name
HAVING AVG(duration) > (SELECT AVG(duration) FROM Calls);
```

2021/1/22

1532. The Most Recent Three Orders

Table: Customers

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| customer_id   | int     |
| name          | varchar |
+---------------+---------+
```

customer_id is the primary key for this table.

This table contains information about customers.

Table: Orders

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| order_id      | int     |
| order_date    | date    |
| customer_id   | int     |
| cost          | int     |
+---------------+---------+
```

order_id is the primary key for this table.

This table contains information about the orders made by customer_id.

Each customer has **one order per day**.

Write an SQL query to find the most recent 3 orders of each user. If a user ordered less than 3 orders return all of their orders.

Return the result table sorted by customer_name in **ascending** order and in case of a tie by the customer_id in **ascending** order. If there still a tie, order them by the order_date in **descending** order.

The query result format is in the following example:

Customers

```
+-------------+-----------+
| customer_id | name      |
+-------------+-----------+
| 1           | Winston   |
| 2           | Jonathan  |
| 3           | Annabelle |
| 4           | Marwan    |
| 5           | Khaled    |
+-------------+-----------+
```

Orders

```
+----------+------------+-------------+------+
| order_id | order_date | customer_id | cost |
+----------+------------+-------------+------+
| 1        | 2020-07-31 | 1           | 30   |
| 2        | 2020-07-30 | 2           | 40   |
| 3        | 2020-07-31 | 3           | 70   |
| 4        | 2020-07-29 | 4           | 100  |
| 5        | 2020-06-10 | 1           | 1010 |
| 6        | 2020-08-01 | 2           | 102  |
| 7        | 2020-08-01 | 3           | 111  |
| 8        | 2020-08-03 | 1           | 99   |
| 9        | 2020-08-07 | 2           | 32   |
| 10       | 2020-07-15 | 1           | 2    |
+----------+------------+-------------+------+
```

Result table:

```
+---------------+-------------+----------+------------+
| customer_name | customer_id | order_id | order_date |
+---------------+-------------+----------+------------+
| Annabelle     | 3           | 7        | 2020-08-01 |

| Annabelle     | 3           | 3        | 2020-07-31 |

| Jonathan      | 2           | 9        | 2020-08-07 |

| Jonathan      | 2           | 6        | 2020-08-01 |

| Jonathan      | 2           | 2        | 2020-07-30 |

| Marwan        | 4           | 4        | 2020-07-29 |

| Winston       | 1           | 8        | 2020-08-03 |

| Winston       | 1           | 1        | 2020-07-31 |

| Winston       | 1           | 10       | 2020-07-15 |

+---------------+-------------+----------+------------+
```

Winston has 4 orders, we discard the order of "2020-06-10" because it is the oldest order.

Annabelle has only 2 orders, we return them.

Jonathan has exactly 3 orders.

Marwan ordered only one time.

We sort the result table by customer_name in ascending order, by customer_id in ascending order and by order_date in descending order in case of a tie.

```
SELECT a.name AS customer_name, a.customer_id, b.order_id,
b.order_date

FROM Customers a JOIN Orders b ON a.customer_id = b.customer_id

WHERE


(SELECT COUNT(*) FROM Orders c

 WHERE b.customer_id = c.customer_id AND b.order_date < c.order_date)
<= 2


ORDER BY customer_name ASC, customer_id ASC, order_date DESC;
```

1549. The Most Recent Orders for Each Product

Table: Customers

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| customer_id   | int     |
| name          | varchar |
+---------------+---------+
```

customer_id is the primary key for this table.

This table contains information about the customers.

Table: Orders

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| order_id      | int     |
| order_date    | date    |
| customer_id   | int     |
| product_id    | int     |
+---------------+---------+
```

order_id is the primary key for this table.

This table contains information about the orders made by customer_id.

There will be no product ordered by the same user **more than once** in one day.

Table: Products

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| product_id    | int     |
| product_name  | varchar |
| price         | int     |
+---------------+---------+
```

product_id is the primary key for this table.

This table contains information about the Products.

Write an SQL query to find the most recent order(s) of each product.

Return the result table sorted by product_name in **ascending** order and in case of a tie by the product_id in **ascending** order. If there still a tie, order them by the order_id in **ascending** order.

The query result format is in the following example:

Customers

```
+-------------+-----------+
| customer_id | name      |
+-------------+-----------+
| 1           | Winston   |
| 2           | Jonathan  |
| 3           | Annabelle |
| 4           | Marwan    |
| 5           | Khaled    |
+-------------+-----------+
```

Orders

```
+----------+------------+-------------+------------+
| order_id | order_date | customer_id | product_id |
+----------+------------+-------------+------------+
| 1        | 2020-07-31 | 1           | 1          |
| 2        | 2020-07-30 | 2           | 2          |
| 3        | 2020-08-29 | 3           | 3          |
| 4        | 2020-07-29 | 4           | 1          |
| 5        | 2020-06-10 | 1           | 2          |
| 6        | 2020-08-01 | 2           | 1          |
| 7        | 2020-08-01 | 3           | 1          |
| 8        | 2020-08-03 | 1           | 2          |
| 9        | 2020-08-07 | 2           | 3          |
| 10       | 2020-07-15 | 1           | 2          |
+----------+------------+-------------+------------+
```

Products

```
+------------+--------------+-------+
| product_id | product_name | price |
+------------+--------------+-------+
| 1          | keyboard     | 120   |
| 2          | mouse        | 80    |
| 3          | screen       | 600   |
| 4          | hard disk    | 450   |
+------------+--------------+-------+
```

Result table:

```
+--------------+------------+----------+------------+
| product_name | product_id | order_id | order_date |
+--------------+------------+----------+------------+
| keyboard     | 1          | 6        | 2020-08-01 |
| keyboard     | 1          | 7        | 2020-08-01 |
| mouse        | 2          | 8        | 2020-08-03 |
| screen       | 3          | 3        | 2020-08-29 |
+--------------+------------+----------+------------+
```

keyboard's most recent order is in 2020-08-01, it was ordered two times this day.

mouse's most recent order is in 2020-08-03, it was ordered only once this day.

screen's most recent order is in 2020-08-29, it was ordered only once this day.

The hard disk was never ordered and we don't include it in the result table.

```
SELECT b.product_name, a.product_id, a.order_id, a.order_date
FROM Orders a JOIN Products b ON a.product_id = b.product_id
WHERE
 (a.product_id, a.order_date) IN
 (SELECT product_id, MAX(order_date) AS order_date
  FROM Orders
  GROUP BY product_id)
ORDER BY b.product_name, a.product_id, a.order_id;
```

1555. Bank Account Summary

Table: Users

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| user_id      | int     |
| user_name    | varchar |
| credit       | int     |
+--------------+---------+
```

user_id is the primary key for this table.

Each row of this table contains the current credit information for each user.

Table: Transactions

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| trans_id      | int     |
| paid_by       | int     |
| paid_to       | int     |
| amount        | int     |
| transacted_on | date    |
+---------------+---------+
```

trans_id is the primary key for this table.

Each row of this table contains the information about the transaction in the bank.

User with id (paid_by) transfer money to user with id (paid_to).

Leetcode Bank (LCB) helps its coders in making virtual payments. Our bank records all transactions in the table *Transaction*, we want to find out the current balance of all users and check wheter they have breached their credit limit (If their current credit is less than 0).

Write an SQL query to report.

- user_id

- user_name
- credit, current balance after performing transactions.
- credit_limit_breached, check credit_limit ("Yes" or "No")

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

The query result format is in the following example.

Users table:

```
+------------+--------------+-------------+
| user_id    | user_name    | credit      |
+------------+--------------+-------------+
| 1          | Moustafa     | 100         |
| 2          | Jonathan     | 200         |
| 3          | Winston      | 10000       |
| 4          | Luis         | 800         |
+------------+--------------+-------------+
```

Transactions table:

```
+------------+------------+------------+----------+---------------+
| trans_id   | paid_by    | paid_to    | amount   | transacted_on |
+------------+------------+------------+----------+---------------+
| 1          | 1          | 3          | 400      | 2020-08-01    |
| 2          | 3          | 2          | 500      | 2020-08-02    |
| 3          | 2          | 1          | 200      | 2020-08-03    |
+------------+------------+------------+----------+---------------+
```

Result table:

```
+------------+------------+------------+-----------------------+
| user_id    | user_name  | credit     | credit_limit_breached |
+------------+------------+------------+-----------------------+
| 1          | Moustafa   | -100       | Yes                   |
| 2          | Jonathan   | 500        | No                    |
| 3          | Winston    | 9900       | No                    |
| 4          | Luis       | 800        | No                    |
+------------+------------+------------+-----------------------+
```

Moustafa paid $400 on "2020-08-01" and received $200 on "2020-08-03", credit (100 -400 +200) = -$100

Jonathan received $500 on "2020-08-02" and paid $200 on "2020-08-08", credit (200 +500 -200) = $500

Winston received $400 on "2020-08-01" and paid $500 on "2020-08-03", credit (10000 +400 -500) = $9990

Luis didn't received any transfer, credit = $800

```
SELECT U.user_id,

        user_name,

        (credit - out_cash + in_cash) AS credit,

        IF((credit - out_cash + in_cash) < 0, 'Yes', 'No') AS credit_limit_breached

FROM Users U

JOIN

        (SELECT U1.user_id,

        IFNULL(SUM(amount), 0) AS out_cash

        FROM Users U1 LEFT JOIN transactions T1 ON U1.user_id = T1.paid_by

        GROUP BY user_id) out_tmp ON U.user_id = out_tmp.user_id

JOIN

        (SELECT U2.user_id,

        IFNULL(SUM(amount), 0) AS in_cash

        FROM Users U2 LEFT JOIN transactions T2 ON U2.user_id = T2.paid_to

        GROUP BY user_id) in_tmp ON U.user_id = in_tmp.user_id;
```

2021/1/24

1596. The Most Frequently Ordered Products for Each Customer

```
SELECT customer_id, product_id, product_name

FROM


(SELECT O.customer_id, O.product_id, P.product_name,

 RANK() OVER (PARTITION BY customer_id ORDER BY COUNT(O.product_id) DESC) AS rnk

 FROM Orders O LEFT JOIN Products P ON O.product_id = P.product_id

 GROUP BY customer_id, product_id) AS t


WHERE rnk = 1;
```

1613. Find the Missing IDs

Table: Customers

```
+---------------+---------+

| Column Name   | Type    |

+---------------+---------+

| customer_id   | int     |

| customer_name | varchar |

+---------------+---------+
```

customer_id is the primary key for this table.

Each row of this table contains the name and the id customer.

Write an SQL query to find the missing customer IDs. The missing IDs are ones that are not in the Customers table but are in the range between 1 and the **maximum** customer_id present in the table.

Notice that the maximum customer_id will not exceed 100.

Return the result table ordered by ids in **ascending order**.

The query result format is in the following example.

Customers table:

```
+-------------+---------------+
| customer_id | customer_name |
+-------------+---------------+
| 1           | Alice         |
| 4           | Bob           |
| 5           | Charlie       |
+-------------+---------------+
```

Result table:

```
+-----+
| ids |
+-----+
| 2   |
| 3   |
+-----+
```

The maximum customer_id present in the table is 5, so in the range [1,5], IDs 2 and 3 are missing from the table.

```
WITH RECURSIVE id_seq AS
(SELECT 1 AS continued_id UNION SELECT continued_id + 1
 FROM id_seq
 WHERE continued_id < (SELECT MAX(customer_id) FROM Customers))

SELECT continued_id AS ids
FROM id_seq
WHERE continued_id NOT IN (SELECT customer_id FROM Customers);
```

1699. Number of Calls Between Two Persons

Table: Calls

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| from_id     | int     |
| to_id       | int     |
| duration    | int     |
+-------------+---------+
```

This table does not have a primary key, it may contain duplicates.

This table contains the duration of a phone call between from_id and to_id.

from_id != to_id

Write an SQL query to report the number of calls and the total call duration between each pair of distinct persons (person1, person2) where person1 < person2.

Return the result table in any order.

The query result format is in the following example:

Calls table:

```
+---------+-------+----------+
| from_id | to_id | duration |
+---------+-------+----------+
| 1       | 2     | 59       |
| 2       | 1     | 11       |
| 1       | 3     | 20       |

| 3       | 4     | 100      |

| 3       | 4     | 200      |

| 3       | 4     | 200      |

| 4       | 3     | 499      |

+---------+-------+----------+
```

Result table:

```
+---------+---------+------------+----------------+

| person1 | person2 | call_count | total_duration |

+---------+---------+------------+----------------+

| 1       | 2       | 2          | 70             |

| 1       | 3       | 1          | 20             |

| 3       | 4       | 4          | 999            |

+---------+---------+------------+----------------+
```

Users 1 and 2 had 2 calls and the total duration is 70 (59 + 11).

Users 1 and 3 had 1 call and the total duration is 20.

Users 3 and 4 had 4 calls and the total duration is 999 (100 + 200 + 200 + 499).

```
SELECT LEAST(from_id,to_id) AS person1,
        GREATEST(from_id,to_id) AS person2,
        COUNT(*) AS call_count,
        SUM(duration) AS total_duration
FROM Calls
GROUP BY person1, person2;
```

```
SELECT CASE WHEN from_id > to_id THEN to_id
        ELSE from_id
        END AS person1,
        CASE WHEN from_id > to_id THEN from_id
        ELSE to_id
        END AS person2,
        COUNT(duration) AS call_count,
        SUM(duration) AS total_duration
FROM Calls
GROUP BY person1, person2;
```

1709. Biggest Window Between Visits

Table: UserVisits

```
+-------------+------+

| Column Name | Type |

+-------------+------+

| user_id     | int  |

| visit_date  | date |

+-------------+------+
```

This table does not have a primary key.

This table contains logs of the dates that users vistied a certain retailer.

Assume today's date is '2021-1-1'.

Write an SQL query that will, for each user_id, find out the largest window of days between each visit and the one right after it (or today if you are considering the last visit).

Return the result table ordered by user_id.

The query result format is in the following example:

UserVisits table:

```
+---------+------------+

| user_id | visit_date |
+---------+------------+
| 1       | 2020-11-28 |
| 1       | 2020-10-20 |
| 1       | 2020-12-3  |
| 2       | 2020-10-5  |
| 2       | 2020-12-9  |
| 3       | 2020-11-11 |
+---------+------------+
```

Result table:

```
+---------+---------------+
| user_id | biggest_window|
+---------+---------------+
| 1       | 39            |
| 2       | 65            |
| 3       | 51            |
+---------+---------------+
```

For the first user, the windows in question are between dates:

- 2020-10-20 and 2020-11-28 with a total of 39 days.
- 2020-11-28 and 2020-12-3 with a total of 5 days.
- 2020-12-3 and 2021-1-1 with a total of 29 days.

Making the biggest window the one with 39 days.

For the second user, the windows in question are between dates:

- 2020-10-5 and 2020-12-9 with a total of 65 days.
- 2020-12-9 and 2021-1-1 with a total of 23 days.

Making the biggest window the one with 65 days.

For the third user, the only window in question is between dates 2020-11-11 and 2021-1-1 with a total of 51 days.

```
SELECT user_id, MAX(diff) AS biggest_window

FROM

(SELECT user_id, DATEDIFF(LEAD(visit_date, 1, '2021-01-01') OVER
(PARTITION BY user_id ORDER BY visit_date), visit_date) AS diff

 FROM userVisits) AS a

GROUP BY user_id

ORDER BY user_id;
```

1715. Count Apples and Oranges

Table: Boxes

```
+--------------+------+

| Column Name  | Type |

+--------------+------+

| box_id       | int  |
| chest_id     | int  |

| apple_count  | int  |

| orange_count | int  |

+--------------+------+
```

box_id is the primary key for this table.

chest_id is a foreign key of the chests table.

This table contains information about the boxes and the number of oranges and apples they contain. Each box may contain a chest, which also can contain oranges and apples.

Table: Chests

```
+--------------+------+

| Column Name  | Type |

+--------------+------+

| chest_id     | int  |

| apple_count  | int  |

| orange_count | int  |

+--------------+------+
```

chest_id is the primary key for this table.

This table contains information about the chests we have, and the corresponding number if oranges and apples they contain.

Write an SQL query to count the number of apples and oranges in all the boxes. If a box contains a chest, you should also include the number of apples and oranges it has.

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

The query result format is in the following example:

Boxes table:

```
+--------+----------+-------------+--------------+
| box_id | chest_id | apple_count | orange_count |
+--------+----------+-------------+--------------+
| 2      | null     | 6           | 15           |
| 18     | 14       | 4           | 15           |
| 19     | 3        | 8           | 4            |
| 12     | 2        | 19          | 20           |
| 20     | 6        | 12          | 9            |
| 8      | 6        | 9           | 9            |
| 3      | 14       | 16          | 7            |
+--------+----------+-------------+--------------+
```

Chests table:

```
+----------+-------------+--------------+
| chest_id | apple_count | orange_count |
+----------+-------------+--------------+
| 6        | 5           | 6            |
| 14       | 20          | 10           |
| 2        | 8           | 8            |
| 3        | 19          | 4            |
| 16       | 19          | 19           |
+----------+-------------+--------------+
```

Result table:

```
+-------------+--------------+
| apple_count | orange_count |
+-------------+--------------+
| 151         | 123          |
+-------------+--------------+
```

box 2 has 6 apples and 15 oranges.

box 18 has 4 + 20 (from the chest) = 24 apples and 15 + 10 (from the chest) = 25 oranges.

box 19 has 8 + 19 (from the chest) = 27 apples and 4 + 4 (from the chest) = 8 oranges.

box 12 has 19 + 8 (from the chest) = 27 apples and 20 + 8 (from the chest) = 28 oranges.

box 20 has 12 + 5 (from the chest) = 17 apples and 9 + 6 (from the chest) = 15 oranges.

box 8 has 9 + 5 (from the chest) = 14 apples and 9 + 6 (from the chest) = 15 oranges.

box 3 has 16 + 20 (from the chest) = 36 apples and 7 + 10 (from the chest) = 17 oranges.

Total number of apples = 6 + 24 + 27 + 27 + 17 + 14 + 36 = 151

Total number of oranges = 15 + 25 + 8 + 28 + 15 + 15 + 17 = 123

```
SELECT SUM(CASE WHEN b.chest_id IS NOT NULL THEN (b.apple_count + c.apple_count)
        ELSE b.apple_count END) AS apple_count,
        SUM(CASE WHEN b.chest_id IS NOT NULL THEN (b.orange_count + c.orange_count)
        ELSE b.orange_count END) AS orange_count
FROM boxes b LEFT JOIN chests c ON b.chest_id = c.chest_id;
```