# My LeetCode Submissions - @Sm8fkGnEZ7

Download PDF | Follow @TheShubham99 on GitHub | Star on GitHub | View Source Code

## 185 Department Top Three Salaries (link)

### Description

Table: `Employee`

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| id           | int     |
| name         | varchar |
| salary       | int     |
| departmentId | int     |
+--------------+---------+
id is the primary key (column with unique values) for this table.
departmentId is a foreign key (reference column) of the ID from the Department table.
Each row of this table indicates the ID, name, and salary of an employee. It also contains the ID of their department.
```

Table: `Department`

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

```
id is the primary key (column with unique values) for this table.
Each row of this table indicates the ID of a department and its name.
```

A company's executives are interested in seeing who earns the most money in each of the company's departments. A **high earner** in a department is an employee who has a salary in the **top three unique** salaries for that department.

Write a solution to find the employees who are **high earners** in each of the departments.

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

The result format is in the following example.

### Example 1:

```
Input:
Employee table:
+----+-------+--------+--------------+
| id | name  | salary | departmentId |
+----+-------+--------+--------------+
| 1  | Joe   | 85000  | 1            |
| 2  | Henry | 80000  | 2            |
| 3  | Sam   | 60000  | 2            |
| 4  | Max   | 90000  | 1            |
| 5  | Janet | 69000  | 1            |
| 6  | Randy | 85000  | 1            |
| 7  | Will  | 70000  | 1            |
+----+-------+--------+--------------+
Department table:
+----+-------+
| id | name  |
+----+-------+
| 1  | IT    |
| 2  | Sales |
+----+-------+
Output:
+------------+----------+--------+
| Department | Employee | Salary |
+------------+----------+--------+
| IT         | Max      | 90000  |
| IT         | Joe      | 85000  |
```

```
| IT         | Randy    | 85000  |
| IT         | Will     | 70000  |
| Sales      | Henry    | 80000  |
| Sales      | Sam      | 60000  |
+------------+----------+--------+
Explanation:
In the IT department:
- Max earns the highest unique salary
- Both Randy and Joe earn the second-highest unique salary
- Will earns the third-highest unique salary

In the Sales department:
- Henry earns the highest salary
- Sam earns the second-highest salary
- There is no third-highest salary as there are only two employees
```

### Constraints:

- There are no employees with the **exact** same name, salary *and* department.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT
    d.name AS Department,
    e.name AS Employee,
    e.salary AS Salary
FROM (
    SELECT *,
           DENSE_RANK() OVER (
               PARTITION BY departmentId
               ORDER BY salary DESC
           ) AS salary_rank
    FROM Employee
) e
JOIN Department d
ON e.departmentId = d.id
WHERE e.salary_rank <= 3;
```

# 585 Investments in 2016 (link)

## Description

Table: `Insurance`

```
+-------------+-------+
| Column Name | Type  |
+-------------+-------+
| pid         | int   |
| tiv_2015    | float |
| tiv_2016    | float |
| lat         | float |
| lon         | float |
+-------------+-------+
pid is the primary key (column with unique values) for this table.
Each row of this table contains information about one policy where:
pid is the policyholder's policy ID.
tiv_2015 is the total investment value in 2015 and tiv_2016 is the total investment value in 2016.
lat is the latitude of the policy holder's city. It's guaranteed that lat is not NULL.
lon is the longitude of the policy holder's city. It's guaranteed that lon is not NULL.
```

Write a solution to report the sum of all total investment values in 2016 `tiv_2016`, for all policyholders who:

- have the same `tiv_2015` value as one or more other policyholders, and
- are not located in the same city as any other policyholder (i.e., the (`lat, lon`) attribute pairs must be unique).

Round `tiv_2016` to **two decimal places**.

The result format is in the following example.

**Example 1:**

```
Input:
Insurance table:
```

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

**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 the same as the third and fourth records, and its location is 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 sa
So, the result is the sum of tiv_2016 of the first and last record, which is 45.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT
    ROUND(SUM(tiv_2016), 2) AS tiv_2016
FROM Insurance
WHERE tiv_2015 IN (
    SELECT tiv_2015
    FROM Insurance
    GROUP BY tiv_2015
    HAVING COUNT(*) > 1
)
AND (lat, lon) IN (
    SELECT lat, lon
    FROM Insurance
    GROUP BY lat, lon
    HAVING COUNT(*) = 1
);
```

# 602 Friend Requests II: Who Has the Most Friends (link)

## Description

Table: `RequestAccepted`

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| requester_id   | int     |
| accepter_id    | int     |
| accept_date    | date    |
+----------------+---------+
(requester_id, accepter_id) is the primary key (combination of columns with unique values) for this table.
This table contains the ID of the user who sent the request, the ID of the user who received the request, and the date when the requ
```

Write a solution to find the people who have the most friends and the most friends number.

The test cases are generated so that only one person has the most friends.

The result format is in the following example.

**Example 1:**

```
Input:
RequestAccepted table:
+--------------+-------------+-------------+
| 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  |
+--------------+-------------+-------------+
```

**Output:**

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

**Explanation:**

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

**Follow up:** In the real world, multiple people could have the same most number of friends. Could you find all these people in this case?

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT id, COUNT(*) AS num
FROM (
    SELECT requester_id  AS id FROM RequestAccepted
    UNION ALL
    SELECT accepter_id AS id FROM RequestAccepted
) f
GROUP BY id
HAVING COUNT(*) = (
    SELECT MAX(cnt)
    FROM (
        SELECT COUNT(*) AS cnt
        FROM (
            SELECT requester_id AS id FROM RequestAccepted
            UNION ALL
            SELECT accepter_id AS id FROM RequestAccepted
        ) x
        GROUP BY id
    ) y
);
```

# 1327 Last Person to Fit in the Bus (link)

## Description

Table: `Queue`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| person_id   | int     |
| person_name | varchar |
| weight      | int     |
| turn        | int     |
+-------------+---------+
person_id column contains unique values.
This table has the information about all people waiting for a bus.
The person_id and turn columns will contain all numbers from 1 to n, where n is the number of rows in the table.
turn determines the order of which the people will board the bus, where turn=1 denotes the first person to board and turn=n denotes
weight is the weight of the person in kilograms.
```

There is a queue of people waiting to board a bus. However, the bus has a weight limit of `1000` **kilograms**, so there may be some people who cannot board.

Write a solution to find the `person_name` of the **last person** that can fit on the bus without exceeding the weight limit. The test cases are generated such that the first person does not exceed the weight limit.

**Note** that *only one* person can board the bus at any given turn.

The result format is in the following example.

**Example 1:**

**Input:**
Queue table:

| person_id | person_name | weight | turn |
|---|---|---|---|
| 5 | Alice | 250 | 1 |
| 4 | Bob | 175 | 5 |
| 3 | Alex | 350 | 2 |
| 6 | John Cena | 400 | 3 |
| 1 | Winston | 500 | 6 |
| 2 | Marie | 200 | 4 |

**Output:**

| person_name |
|---|
| John Cena |

**Explanation:** The folowing table is ordered by the turn for simplicity.

| Turn | ID | Name | Weight | Total Weight | |
|---|---|---|---|---|---|
| 1 | 5 | Alice | 250 | 250 | |
| 2 | 3 | Alex | 350 | 600 | |
| 3 | 6 | John Cena | 400 | 1000 | (last person to board) |
| 4 | 2 | Marie | 200 | 1200 | (cannot board) |
| 5 | 4 | Bob | 175 | ___ | |
| 6 | 1 | Winston | 500 | ___ | |

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT person_name
FROM (
    SELECT
        person_name,
        SUM(weight) OVER (ORDER BY turn) AS total_weight
    FROM Queue
) q
WHERE total_weight = (
    SELECT MAX(total_weight)
    FROM (
        SELECT SUM(weight) OVER (ORDER BY turn) AS total_weight
        FROM Queue
    ) t
    WHERE total_weight <= 1000
);
```

# 1452 Restaurant Growth (link)

## Description

Table: `Customer`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| customer_id   | int     |
| name          | varchar |
| visited_on    | date    |
| amount        | int     |
+---------------+---------+
In SQL,(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) has 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).

Compute the moving average of how much the customer paid in a seven days window (i.e., current day + 6 days before). `average_amount` should be **rounded to two decimal places**.

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

The result format is in the following example.

**Example 1:**

```
Input:
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         |
+-------------+--------------+--------------+-------------+
```

**Output:**

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

**Explanation:**

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

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
WITH daily AS (
    SELECT visited_on, SUM(amount) AS amount
    FROM Customer
    GROUP BY visited_on
)
SELECT
    d1.visited_on,
    SUM(d2.amount) AS amount,
    ROUND(SUM(d2.amount) / 7, 2) AS average_amount
FROM daily d1
JOIN daily d2
  ON d2.visited_on BETWEEN DATE_SUB(d1.visited_on, INTERVAL 6 DAY) AND d1.visited_on
GROUP BY d1.visited_on
HAVING COUNT(*) = 7
ORDER BY d1.visited_on;
```

# 1480 Movie Rating (link)

## Description

Table: `Movies`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| movie_id      | int     |
| title         | varchar |
+---------------+---------+
movie_id is the primary key (column with unique values) for this table.
title is the name of the movie.
Each movie has a unique title.
```

Table: `Users`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| name          | varchar |
+---------------+---------+
user_id is the primary key (column with unique values) for this table.
The column 'name' has unique values.
```

Table: `MovieRating`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| movie_id      | int     |
| user_id       | int     |
| rating        | int     |
| created_at    | date    |
+---------------+---------+
(movie_id, user_id) is the primary key (column with unique values) 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 a solution to:

- Find the name of the user who has rated the greatest number of movies. In case of a tie, return the lexicographically smaller user name.
- Find the movie name with the **highest average** rating in `February 2020`. In case of a tie, return the lexicographically smaller movie name.

The result format is in the following example.

**Example 1:**

```
Input:
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        |
+-------------+--------------+
MovieRating 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  |
+-------------+--------------+--------------+-------------+
```

**Output:**

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

**Explanation:**

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.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT MIN(name) AS results
FROM (
    SELECT u.name AS name, COUNT(*) AS cnt
    FROM Users u
    JOIN MovieRating r ON r.user_id = u.user_id
    GROUP BY u.user_id, u.name
) t
WHERE t.cnt = (SELECT MAX(cnt) FROM (
    SELECT COUNT(*) AS cnt
    FROM MovieRating
    GROUP BY user_id
) x)

UNION ALL

SELECT MIN(title) AS results
FROM (
    SELECT m.title AS title, AVG(r.rating) AS avg_rating
    FROM Movies m
    JOIN MovieRating r ON r.movie_id = m.movie_id
    WHERE r.created_at BETWEEN '2020-02-01' AND '2020-02-29'
    GROUP BY m.movie_id, m.title
) t2
WHERE t2.avg_rating = (SELECT MAX(avg_rating) FROM (
    SELECT AVG(rating) AS avg_rating
    FROM MovieRating
    WHERE created_at BETWEEN '2020-02-01' AND '2020-02-29'
    GROUP BY movie_id
) y);
```

# 626 Exchange Seats (link)

## Description

Table: `Seat`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| student     | varchar |
+-------------+---------+
id is the primary key (unique value) column for this table.
Each row of this table indicates the name and the ID of a student.
The ID sequence always starts from 1 and increments continuously.
```

Write a solution to swap the seat id of every two consecutive students. If the number of students is odd, the id of the last student is not swapped.

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

The result format is in the following example.

**Example 1:**

```
Input:
Seat table:
+----+---------+
| id | student |
+----+---------+
| 1  | Abbot   |
| 2  | Doris   |
| 3  | Emerson |
| 4  | Green   |
| 5  | Jeames  |
```

```
+----+---------+
```
**Output:**
```
+----+---------+
| id | student |
+----+---------+
| 1  | Doris   |
| 2  | Abbot   |
| 3  | Green   |
| 4  | Emerson |
| 5  | Jeames  |
+----+---------+
```
**Explanation:**
Note that if the number of students is odd, there is no need to change the last one's seat.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

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

# 1664 Find Users With Valid E-Mails (link)

## Description

Table: `Users`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| name          | varchar |
| mail          | varchar |
+---------------+---------+
user_id is the primary key (column with unique values) for this table.
This table contains information of the users signed up in a website. Some e-mails are invalid.
```

Write a solution to find the users who have **valid emails**.

A valid e-mail has a prefix name and a domain where:

- **The prefix name** is a string that may contain letters (upper or lower case), digits, underscore `'_'`, period `'.'`, and/or dash `'-'`. The prefix name **must** start with a letter.
- **The domain** is `'@leetcode.com'`.

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

The result format is in the following example.

**Example 1:**

```
Input:
Users table:
+---------+-----------+-------------------------+
| user_id | name      | mail                    |
```

```
+---------+-----------+-------------------------+
| 1       | Winston   | winston@leetcode.com    |
| 2       | Jonathan  | jonathanisgreat         |
| 3       | Annabelle | bella-@leetcode.com     |
| 4       | Sally     | sally.come@leetcode.com |
| 5       | Marwan    | quarz#2020@leetcode.com |
| 6       | David     | david69@gmail.com       |
| 7       | Shapiro   | .shapo@leetcode.com     |
+---------+-----------+-------------------------+
```

**Output:**

```
+---------+-----------+-------------------------+
| user_id | name      | mail                    |
+---------+-----------+-------------------------+
| 1       | Winston   | winston@leetcode.com    |
| 3       | Annabelle | bella-@leetcode.com     |
| 4       | Sally     | sally.come@leetcode.com |
+---------+-----------+-------------------------+
```

**Explanation:**

The mail of user 2 does not have a domain.
The mail of user 5 has the # sign which is not allowed.
The mail of user 6 does not have the leetcode domain.
The mail of user 7 starts with a period.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT user_id, name, mail
FROM Users
WHERE
    BINARY RIGHT(mail, 13) = '@leetcode.com'
    AND (LENGTH(mail) - LENGTH(REPLACE(mail, '@', ''))) = 1
    AND LEFT(mail, LOCATE('@', mail) - 1) REGEXP '^[A-Za-z][A-Za-z0-9._-]*$';
```

# 1462 List the Products Ordered in a Period (link)

## Description

Table: `Products`

```
+------------------+---------+
| Column Name      | Type    |
+------------------+---------+
| product_id       | int     |
| product_name     | varchar |
| product_category | varchar |
+------------------+---------+
product_id is the primary key (column with unique values) for this table.
This table contains data about the company's products.
```

Table: `Orders`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| product_id    | int     |
| order_date    | date    |
| unit          | int     |
+---------------+---------+
This table may have duplicate rows.
product_id is a foreign key (reference column) to the Products table.
unit is the number of products ordered in order_date.
```

Write a solution to get the names of products that have at least `100` units ordered in **February 2020** and their amount.

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

The result format is in the following example.

### Example 1:

**Input:**
Products table:

| product_id | product_name | product_category |
|---|---|---|
| 1 | Leetcode Solutions | Book |
| 2 | Jewels of Stringology | Book |
| 3 | HP | Laptop |
| 4 | Lenovo | Laptop |
| 5 | Leetcode Kit | T-shirt |

Orders table:

| product_id | order_date | unit |
|---|---|---|
| 1 | 2020-02-05 | 60 |
| 1 | 2020-02-10 | 70 |
| 2 | 2020-01-18 | 30 |
| 2 | 2020-02-11 | 80 |
| 3 | 2020-02-17 | 2 |
| 3 | 2020-02-24 | 3 |
| 4 | 2020-03-01 | 20 |
| 4 | 2020-03-04 | 30 |
| 4 | 2020-03-04 | 60 |
| 5 | 2020-02-25 | 50 |
| 5 | 2020-02-27 | 50 |
| 5 | 2020-03-01 | 50 |

**Output:**

| product_name | unit |
|---|---|
| Leetcode Solutions | 130 |
| Leetcode Kit | 100 |

**Explanation:**
Products with product_id = 1 is ordered in February a total of (60 + 70) = 130.
Products with product_id = 2 is ordered in February a total of 80.
Products with product_id = 3 is ordered in February a total of (2 + 3) = 5.
Products with product_id = 4 was not ordered in February 2020.
Products with product_id = 5 is ordered in February a total of (50 + 50) = 100.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT p.product_name, SUM(o.unit) AS unit
FROM Products p
JOIN Orders o
ON p.product_id = o.product_id
WHERE o.order_date >= '2020-02-01'
  AND o.order_date <  '2020-03-01'
GROUP BY p.product_id, p.product_name
HAVING SUM(o.unit) >= 100;
```

# 1625 Group Sold Products By The Date (link)

## Description

Table `Activities`:

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| sell_date   | date    |
| product     | varchar |
+-------------+---------+
There is no primary key (column with unique values) for this table. It may contain duplicates.
Each row of this table contains the product name and the date it was sold in a market.
```

Write a solution to find for each date the number of different products sold and their names.

The sold products names for each date should be sorted lexicographically.

Return the result table ordered by `sell_date`.

The result format is in the following example.

**Example 1:**

```
Input:
Activities table:
+------------+------------+
| sell_date  | product     |
+------------+------------+
| 2020-05-30 | Headphone  |
| 2020-06-01 | Pencil     |
| 2020-06-02 | Mask       |
| 2020-05-30 | Basketball |
| 2020-06-01 | Bible      |
```

```
| 2020-06-02 | Mask       |
| 2020-05-30 | T-Shirt    |
+------------+------------+
```
**Output:**
```
+------------+----------+-----------------------------+
| sell_date  | num_sold | products                    |
+------------+----------+-----------------------------+
| 2020-05-30 | 3        | Basketball,Headphone,T-shirt |
| 2020-06-01 | 2        | Bible,Pencil                |
| 2020-06-02 | 1        | Mask                        |
+------------+----------+-----------------------------+
```
**Explanation:**
For 2020-05-30, Sold items were (Headphone, Basketball, T-shirt), we sort them lexicographically and separate them by a comma.
For 2020-06-01, Sold items were (Pencil, Bible), we sort them lexicographically and separate them by a comma.
For 2020-06-02, the Sold item is (Mask), we just return it.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT
    sell_date,
    COUNT(DISTINCT product) AS num_sold,
    GROUP_CONCAT(DISTINCT product ORDER BY product ASC) AS products
FROM Activities
GROUP BY sell_date
ORDER BY sell_date;
```

# 176 Second Highest Salary (link)

## Description

Table: `Employee`

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| id          | int  |
| salary      | int  |
+-------------+------+
id is the primary key (column with unique values) for this table.
Each row of this table contains information about the salary of an employee.
```

Write a solution to find the second highest **distinct** salary from the `Employee` table. If there is no second highest salary, return `null` (`return None in Pandas`).

The result format is in the following example.

**Example 1:**

```
Input:
Employee table:
+----+--------+
| id | salary |
+----+--------+
| 1  | 100    |
| 2  | 200    |
| 3  | 300    |
+----+--------+
Output:
+---------------------+
| SecondHighestSalary |
+---------------------+
```

```
| 200                 |
+---------------------+
```

### Example 2:

```
Input:
Employee table:
+----+--------+
| id | salary |
+----+--------+
| 1  | 100    |
+----+--------+
Output:
+---------------------+
| SecondHighestSalary |
+---------------------+
| null                |
+---------------------+
```

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT DISTINCT MAX(salary) as SecondHighestSalary
FROM Employee
WHERE salary < (SELECT DISTINCT MAX(salary)
FROM Employee
);
```

# 196 Delete Duplicate Emails (link)

## Description

Table: `Person`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| email       | varchar |
+-------------+---------+
id is the primary key (column with unique values) for this table.
Each row of this table contains an email. The emails will not contain uppercase letters.
```

Write a solution to **delete** all duplicate emails, keeping only one unique email with the smallest `id`.

For SQL users, please note that you are supposed to write a `DELETE` statement and not a `SELECT` one.

For Pandas users, please note that you are supposed to modify `Person` in place.

After running your script, the answer shown is the `Person` table. The driver will first compile and run your piece of code and then show the `Person` table. The final order of the `Person` table **does not matter**.

The result format is in the following example.

**Example 1:**

```
Input:
Person table:
+----+------------------+
| id | email            |
+----+------------------+
| 1  | john@example.com |
```

```
| 2  | bob@example.com  |
| 3  | john@example.com |
+----+------------------+
```
**Output:**
```
+----+------------------+
| id | email            |
+----+------------------+
| 1  | john@example.com |
| 2  | bob@example.com  |
+----+------------------+
```
**Explanation:** john@example.com is repeated two times. We keep the row with the smallest Id = 1.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
DELETE FROM Person
WHERE id NOT IN (
    SELECT id FROM (
        SELECT MIN(id) AS id
        FROM Person
        GROUP BY email
    ) t
);
```

# 1670 Patients With a Condition (link)

## Description

Table: `Patients`

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| patient_id   | int     |
| patient_name | varchar |
| conditions   | varchar |
+--------------+---------+
patient_id is the primary key (column with unique values) for this table.
'conditions' contains 0 or more code separated by spaces.
This table contains information of the patients in the hospital.
```

Write a solution to find the patient_id, patient_name, and conditions of the patients who have Type I Diabetes. Type I Diabetes always starts with `DIAB1` prefix.

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

The result format is in the following example.

**Example 1:**

```
Input:
Patients table:
+------------+--------------+--------------+
| patient_id | patient_name | conditions   |
+------------+--------------+--------------+
| 1          | Daniel       | YFEV COUGH   |
| 2          | Alice        |              |
| 3          | Bob          | DIAB100 MYOP |
| 4          | George       | ACNE DIAB100 |
```

```
| 5          | Alain        | DIAB201      |
+------------+--------------+--------------+
```
**Output:**
```
+------------+--------------+--------------+
| patient_id | patient_name | conditions   |
+------------+--------------+--------------+
| 3          | Bob          | DIAB100 MYOP |
| 4          | George       | ACNE DIAB100 |
+------------+--------------+--------------+
```
**Explanation:** Bob and George both have a condition that starts with DIAB1.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT patient_id, patient_name, conditions
FROM Patients
WHERE conditions LIKE 'DIAB1%'
   OR conditions LIKE '% DIAB1%';
```

# 2127 Employees Whose Manager Left the Company (link)

## Description

Table: `Employees`

```
+-------------+----------+
| Column Name | Type     |
+-------------+----------+
| employee_id | int      |
| name        | varchar  |
| manager_id  | int      |
| salary      | int      |
+-------------+----------+
In SQL, employee_id is the primary key for this table.
This table contains information about the employees, their salary, and the ID of their manager. Some employees do not have a manager
```

Find the IDs of the employees whose salary is strictly less than `$30000` and whose manager left the company. When a manager leaves the company, their information is deleted from the `Employees` table, but the reports still have their `manager_id` set to the manager that left.

Return the result table ordered by `employee_id`.

The result format is in the following example.

**Example 1:**

```
Input:
Employees table:
+-------------+-----------+------------+--------+
| employee_id | name      | manager_id | salary |
+-------------+-----------+------------+--------+
| 3           | Mila      | 9          | 60301  |
| 12          | Antonella | null       | 31000  |
| 13          | Emery     | null       | 67084  |
```

```
| 1           | Kalel     | 11         | 21241  |
| 9           | Mikaela   | null       | 50937  |
| 11          | Joziah    | 6          | 28485  |
+-------------+-----------+------------+--------+
```

**Output:**

```
+-------------+
| employee_id |
+-------------+
| 11          |
+-------------+
```

**Explanation:**
The employees with a salary less than $30000 are 1 (Kalel) and 11 (Joziah).
Kalel's manager is employee 11, who is still in the company (Joziah).
Joziah's manager is employee 6, who left the company because there is no row for employee 6 as it was deleted.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT employee_id
FROM Employees
WHERE salary < 30000
AND manager_id NOT IN (SELECT employee_id FROM Employees)
ORDER BY employee_id;
```

# 2057 Count Salary Categories (link)

## Description

Table: `Accounts`

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| account_id  | int  |
| income      | int  |
+-------------+------+
account_id is the primary key (column with unique values) for this table.
Each row contains information about the monthly income for one bank account.
```

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

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

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

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

The result format is in the following example.

**Example 1:**

```
Input:
Accounts table:
+------------+--------+
| account_id | income |
+------------+--------+
```

```
| 3          | 108939 |
| 2          | 12747  |
| 8          | 87709  |
| 6          | 91796  |
+------------+--------+
```

**Output:**

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

**Explanation:**

```
Low Salary: Account 2.
Average Salary: No accounts.
High Salary: Accounts 3, 6, and 8.
```

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT c.category, COUNT(a.category) AS accounts_count
FROM (
    SELECT 'Low Salary' AS category
    UNION ALL
    SELECT 'Average Salary'
    UNION ALL
    SELECT 'High Salary'
) c
LEFT JOIN (
    SELECT
      CASE
        WHEN income < 20000 THEN 'Low Salary'
        WHEN income BETWEEN 20000 AND 50000 THEN 'Average Salary'
        ELSE 'High Salary'
      END AS category
    FROM Accounts
) a
ON c.category = a.category
GROUP BY c.category;
```

# 1278 Product Price at a Given Date (link)

## Description

Table: `Products`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| product_id    | int     |
| new_price     | int     |
| change_date   | date    |
+---------------+---------+
(product_id, change_date) is the primary key (combination of columns with unique values) of this table.
Each row of this table indicates that the price of some product was changed to a new price at some date.
```

Initially, all products have price 10.

Write a solution to find the prices of all products on the date `2019-08-16`.

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

The result format is in the following example.

**Example 1:**

```
Input:
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  |
```

```
+------------+-----------+-------------+
```
**Output:**
```
+------------+-------+
| product_id | price |
+------------+-------+
| 2          | 50    |
| 1          | 35    |
| 3          | 10    |
+------------+-------+
```

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT base.product_id,
       COALESCE(p.new_price, 10) AS price
FROM (SELECT DISTINCT product_id FROM Products) base
LEFT JOIN (
    SELECT product_id, MAX(change_date) AS max_change_date
    FROM Products
    WHERE change_date <= '2019-08-16'
    GROUP BY product_id
) pro
  ON base.product_id = pro.product_id
LEFT JOIN Products p
  ON p.product_id = pro.product_id
 AND p.change_date = pro.max_change_date;
```

# 1811 Fix Names in a Table (link)

## Description

Table: `Users`

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| user_id        | int     |
| name           | varchar |
+----------------+---------+
user_id is the primary key (column with unique values) for this table.
This table contains the ID and the name of the user. The name consists of only lowercase and uppercase characters.
```

Write a solution to fix the names so that only the first character is uppercase and the rest are lowercase.

Return the result table ordered by `user_id`.

The result format is in the following example.

**Example 1:**

```
Input:
Users table:
+---------+-------+
| user_id | name  |
+---------+-------+
| 1       | aLice |
| 2       | bOB   |
+---------+-------+
Output:
+---------+-------+
| user_id | name  |
+---------+-------+
```

```
| 1       | Alice |
| 2       | Bob   |
+---------+-------+
```

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT user_id, CONCAT(UPPER(LEFT(name, 1)), LOWER(SUBSTRING(name, 2))) AS name
FROM Users
ORDER BY user_id;
```

# 180 Consecutive Numbers (link)

## Description

Table: `Logs`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| num         | varchar |
+-------------+---------+
In SQL, id is the primary key for this table.
id is an autoincrement column starting from 1.
```

Find all numbers that appear at least three times consecutively.

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

The result format is in the following example.

**Example 1:**

```
Input:
Logs table:
+----+-----+
| id | num |
+----+-----+
| 1  | 1   |
| 2  | 1   |
| 3  | 1   |
| 4  | 2   |
| 5  | 1   |
| 6  | 2   |
| 7  | 2   |
```

```
+----+-----+
```
**Output:**
```
+-----------------+
| ConsecutiveNums |
+-----------------+
| 1               |
+-----------------+
```
**Explanation:** 1 is the only number that appears consecutively for at least three times.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT DISTINCT l1.num as ConsecutiveNums
FROM Logs l1
JOIN Logs l2
ON l1.id = l2.id - 1
JOIN Logs l3 ON l2.id = l3.id - 1
WHERE l1.num = l2.num
AND l2.num = l3.num;
```

# 610 Triangle Judgement (link)

## Description

Table: `Triangle`

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| x           | int  |
| y           | int  |
| z           | int  |
+-------------+------+
In SQL, (x, y, z) is the primary key column for this table.
Each row of this table contains the lengths of three line segments.
```

Report for every three line segments whether they can form a triangle.

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

The result format is in the following example.

**Example 1:**

```
Input:
Triangle table:
+----+----+----+
| x  | y  | z  |
+----+----+----+
| 13 | 15 | 30 |
| 10 | 20 | 15 |
+----+----+----+
Output:
+----+----+----+----------+
| x  | y  | z  | triangle |
```

```
+----+----+----+----------+
| 13 | 15 | 30 | No       |
| 10 | 20 | 15 | Yes      |
+----+----+----+----------+
```

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
# Write your MySQL query statement below
SELECT x, y, z, (CASE WHEN x+y > z AND y+z > x AND x+z> y THEN 'Yes' ELSE 'No' END) as triangle
FROM Triangle;
```

# 1135 Customers Who Bought All Products (link)

## Description

Table: `Customer`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| customer_id | int     |
| product_key | int     |
+-------------+---------+
This table may contain duplicates rows.
customer_id is not NULL.
product_key is a foreign key (reference column) to Product table.
```

Table: `Product`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| product_key | int     |
+-------------+---------+
product_key is the primary key (column with unique values) for this table.
```

Write a solution to report the customer ids from the `Customer` table that bought all the products in the `Product` table.

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

The result format is in the following example.

**Example 1:**

**Input:**
Customer table:

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

Product table:

| product_key |
|---|
| 5 |
| 6 |

**Output:**

| customer_id |
|---|
| 1 |
| 3 |

**Explanation:**
The customers who bought all the products (5 and 6) are customers with IDs 1 and 3.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT customer_id
FROM Customer
GROUP BY customer_id
HAVING COUNT(DISTINCT product_key) = (
    SELECT COUNT(*) FROM Product
);
```

# 619 Biggest Single Number (link)

## Description

Table: `MyNumbers`

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| num         | int  |
+-------------+------+
This table may contain duplicates (In other words, there is no primary key for this table in SQL).
Each row of this table contains an integer.
```

A **single number** is a number that appeared only once in the `MyNumbers` table.

Find the largest **single number**. If there is no **single number**, report `null`.

The result format is in the following example.

**Example 1:**

```
Input:
MyNumbers table:
+-----+
| num |
+-----+
| 8   |
| 8   |
| 3   |
| 3   |
| 1   |
| 4   |
| 5   |
| 6   |
```

```
+-----+
```
**Output:**
```
+-----+
| num |
+-----+
| 6   |
+-----+
```
**Explanation:** The single numbers are 1, 4, 5, and 6.
Since 6 is the largest single number, we return it.

### Example 2:

**Input:**
MyNumbers table:
```
+-----+
| num |
+-----+
| 8   |
| 8   |
| 7   |
| 7   |
| 3   |
| 3   |
| 3   |
+-----+
```
**Output:**
```
+------+
| num  |
+------+
| null |
+------+
```
**Explanation:** There are no single numbers in the input table so we return null.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT MAX(num) as num from MyNumbers
WHERE num NOT IN (
    SELECT num from MyNumbers
    GROUP BY num
    HAVING COUNT(*) > 1
);
```

# 1877 Find Followers Count (link)

## Description

Table: `Followers`

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| user_id     | int  |
| follower_id | int  |
+-------------+------+
(user_id, follower_id) is the primary key (combination of columns with unique values) for this table.
This table contains the IDs of a user and a follower in a social media app where the follower follows the user.
```

Write a solution that will, for each user, return the number of followers.

Return the result table ordered by `user_id` in ascending order.

The result format is in the following example.

**Example 1:**

```
Input:
Followers table:
+---------+-------------+
| user_id | follower_id |
+---------+-------------+
| 0       | 1           |
| 1       | 0           |
| 2       | 0           |
| 2       | 1           |
+---------+-------------+
Output:
+---------+----------------+
```

```
| user_id | followers_count|
+---------+----------------+
| 0       | 1              |
| 1       | 1              |
| 2       | 2              |
+---------+----------------+
```

**Explanation:**

```
The followers of 0 are {1}
The followers of 1 are {0}
The followers of 2 are {0,1}
```

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
select user_id, COUNT(follower_id) as followers_count
from Followers
GROUP BY user_id
ORDER BY user_id ASC;
```

# 596 Classes With at Least 5 Students (link)

## Description

Table: `Courses`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| student     | varchar |
| class       | varchar |
+-------------+---------+
(student, class) is the primary key (combination of columns with unique values) for this table.
Each row of this table indicates the name of a student and the class in which they are enrolled.
```

Write a solution to find all the classes that have **at least five students**.

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

The result format is in the following example.

**Example 1:**

```
Input:
Courses table:
+---------+----------+
| student | class    |
+---------+----------+
| A       | Math     |
| B       | English  |
| C       | Math     |
| D       | Biology  |
| E       | Math     |
| F       | Computer |
| G       | Math     |
```

```
| H       | Math     |
| I       | Math     |
+---------+----------+
```
**Output:**
```
+---------+
| class   |
+---------+
| Math    |
+---------+
```
**Explanation:**
- Math has 6 students, so we include it.
- English has 1 student, so we do not include it.
- Biology has 1 student, so we do not include it.
- Computer has 1 student, so we do not include it.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT class from Courses
Group by class
having count(student)>=5;
```

# 1155 Product Sales Analysis III (link)

## Description

Table: `Sales`

```
+-------------+-------+
| Column Name | Type  |
+-------------+-------+
| sale_id     | int   |
| product_id  | int   |
| year        | int   |
| quantity    | int   |
| price       | int   |
+-------------+-------+
(sale_id, year) is the primary key (combination of columns with unique values) of this table.
Each row records a sale of a product in a given year.
A product may have multiple sales entries in the same year.
Note that the per-unit price.
```

Write a solution to find all sales that occurred in the **first year** each product was sold.

- For each `product_id`, identify the earliest `year` it appears in the `Sales` table.
- Return **all** sales entries for that product in that year.

Return a table with the following columns: **product_id**, **first_year**, **quantity,** and **price**.
Return the result in any order.

**Example 1:**

```
Input:
Sales table:
+---------+------------+------+----------+-------+
| sale_id | product_id | year | quantity | price |
+---------+------------+------+----------+-------+
```

```
| 1          | 100           | 2008 | 10          | 5000  |
| 2          | 100           | 2009 | 12          | 5000  |
| 7          | 200           | 2011 | 15          | 9000  |
+------------+---------------+------+-------------+-------+
```

**Output:**

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

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT s.product_id, s.year AS first_year, s.quantity, s.price
FROM Sales s
JOIN (
    SELECT product_id, MIN(year) AS first_year
    FROM Sales
    GROUP BY product_id
) f
ON s.product_id = f.product_id
AND s.year = f.first_year;
```

# 1245 User Activity for the Past 30 Days I (link)

## Description

Table: `Activity`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| session_id    | int     |
| activity_date | date    |
| activity_type | enum    |
+---------------+---------+
This table may have duplicate rows.
The activity_type column is an ENUM (category) of type ('open_session', 'end_session', 'scroll_down', 'send_message').
The table shows the user activities for a social media website.
Note that each session belongs to exactly one user.
```

Write a solution to find the daily active user count for a period of `30` days ending `2019-07-27` inclusively. A user was active on someday if they made at least one activity on that day.

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

The result format is in the following example.

Note: **Any** activity from (`'open_session'`, `'end_session'`, `'scroll_down'`, `'send_message'`) will be considered valid activity for a user to be considered active on a day.

**Example 1:**

```
Input:
Activity table:
+---------+------------+---------------+---------------+
| user_id | session_id | activity_date | activity_type |
```

```
+---------+------------+---------------+---------------+
| 1       | 1          | 2019-07-20    | open_session  |
| 1       | 1          | 2019-07-20    | scroll_down   |
| 1       | 1          | 2019-07-20    | end_session   |
| 2       | 4          | 2019-07-20    | open_session  |
| 2       | 4          | 2019-07-21    | send_message  |
| 2       | 4          | 2019-07-21    | end_session   |
| 3       | 2          | 2019-07-21    | open_session  |
| 3       | 2          | 2019-07-21    | send_message  |
| 3       | 2          | 2019-07-21    | end_session   |
| 4       | 3          | 2019-06-25    | open_session  |
| 4       | 3          | 2019-06-25    | end_session   |
+---------+------------+---------------+---------------+
```

**Output:**

```
+------------+--------------+
| day        | active_users |
+------------+--------------+
| 2019-07-20 | 2            |
| 2019-07-21 | 2            |
+------------+--------------+
```

**Explanation:** Note that we do not care about days with zero active users.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT
    activity_date AS day,
    COUNT(DISTINCT user_id) AS active_users
FROM Activity
WHERE activity_type IN ('open_session', 'end_session', 'scroll_down', 'send_message')
  AND activity_date BETWEEN '2019-06-28' AND '2019-07-27'
GROUP BY activity_date
ORDER BY day;
```

# 2495 Number of Unique Subjects Taught by Each Teacher (link)

## Description

Table: `Teacher`

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| teacher_id  | int  |
| subject_id  | int  |
| dept_id     | int  |
+-------------+------+
(subject_id, dept_id) is the primary key (combinations of columns with unique values) of this table.
Each row in this table indicates that the teacher with teacher_id teaches the subject subject_id in the department dept_id.
```

Write a solution to calculate the number of unique subjects each teacher teaches in the university.

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

The result format is shown in the following example.

**Example 1:**

```
Input:
Teacher table:
+------------+------------+---------+
| teacher_id | subject_id | dept_id |
+------------+------------+---------+
| 1          | 2          | 3       |
| 1          | 2          | 4       |
| 1          | 3          | 3       |
| 2          | 1          | 1       |
| 2          | 2          | 1       |
| 2          | 3          | 1       |
```

```
| 2          | 4          | 1       |
+------------+------------+---------+
```
**Output:**
```
+------------+-----+
| teacher_id | cnt |
+------------+-----+
| 1          | 2   |
| 2          | 4   |
+------------+-----+
```
**Explanation:**
```
Teacher 1:
  - They teach subject 2 in departments 3 and 4.
  - They teach subject 3 in department 3.
Teacher 2:
  - They teach subject 1 in department 1.
  - They teach subject 2 in department 1.
  - They teach subject 3 in department 1.
  - They teach subject 4 in department 1.
```

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
select teacher_id, COUNT(DISTINCT subject_id) as cnt
from Teacher
group by teacher_id;
```

# 1182 Game Play Analysis IV (link)

## Description

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 (combination of columns with unique values) of this table.
This table shows the activity of players of some games.
Each row is a record of a player who logged in and played a number of games (possibly 0) before logging out on someday using some de
```

Write a solution to report 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 determine the number of players who logged in on the day immediately following their initial login, and divide it by the number of total players.

The result format is in the following example.

**Example 1:**

```
Input:
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            |
```

```
+-----------+-----------+------------+--------------+
```
**Output:**
```
+-----------+
| fraction  |
+-----------+
| 0.33      |
+-----------+
```
**Explanation:**
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

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT ROUND (
    COUNT(DISTINCT a.player_id) * 1.0/
    (SELECT COUNT(DISTINCT player_id) from Activity),
    2)
    AS fraction
from Activity a
JOIN
(
    SELECT player_id, MIN(event_date) as first_login
    from Activity
    group by player_id
) f
on a.player_id = f.player_id
and a.event_date = DATE_ADD(f.first_login, INTERVAL 1 DAY);
```

# 1292 Immediate Food Delivery II (link)

## Description

Table: `Delivery`

```
+-----------------------------+---------+
| Column Name                 | Type    |
+-----------------------------+---------+
| delivery_id                 | int     |
| customer_id                 | int     |
| order_date                  | date    |
| customer_pref_delivery_date | date    |
+-----------------------------+---------+
delivery_id is the column of unique values of this table.
The table holds information about food delivery to customers that make orders at some date and specify a preferred delivery date (or
```

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

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

Write a solution to find the percentage of immediate orders in the first orders of all customers, **rounded to 2 decimal places**.

The result format is in the following example.

**Example 1:**

```
Input:
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                  |
+-------------+-------------+------------+-----------------------------+
```

**Output:**

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

**Explanation:**

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.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT
    ROUND(
        SUM(CASE
                WHEN order_date = customer_pref_delivery_date THEN 1
                ELSE 0
            END
        ) * 100.0 / COUNT(order_date),
        2
    ) AS immediate_percentage
FROM Delivery
WHERE (customer_id, order_date) IN (
    SELECT customer_id, MIN(order_date)
    FROM Delivery
    GROUP BY customer_id
);
```

# 1317 Monthly Transactions I (link)

## Description

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.

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

The query result format is in the following example.

**Example 1:**

```
Input:
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 |
+------+---------+----------+--------+------------+
```

**Output:**

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

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
# Write your MySQL query statement below
select DATE_FORMAT(trans_date, '%Y-%m') AS month, country, COUNT(trans_date) as trans_count, SUM(
 (CASE WHEN state='approved' THEN 1 ELSE 0 END)
) 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;
```

# 1338 Queries Quality and Percentage (link)

## Description

Table: `Queries`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| query_name  | varchar |
| result      | varchar |
| position    | int     |
| rating      | int     |
+-------------+---------+
This table may have duplicate rows.
This table contains information collected from some queries on a database.
The position column has a value from 1 to 500.
The rating column has a value from 1 to 5. Query with rating less than 3 is a poor query.
```

We define query `quality` as:

> The average of the ratio between query rating and its position.

We also define `poor query percentage` as:

> The percentage of all queries with rating less than 3.

Write a solution to find each `query_name`, the `quality` and `poor_query_percentage`.

Both `quality` and `poor_query_percentage` should be **rounded to 2 decimal places**.

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

The result format is in the following example.

### Example 1:

**Input:**
Queries table:

| query_name | result | position | rating |
|---|---|---|---|
| Dog | Golden Retriever | 1 | 5 |
| Dog | German Shepherd | 2 | 5 |
| Dog | Mule | 200 | 1 |
| Cat | Shirazi | 5 | 2 |
| Cat | Siamese | 3 | 3 |
| Cat | Sphynx | 7 | 4 |

**Output:**

| query_name | quality | poor_query_percentage |
|---|---|---|
| Dog | 2.50 | 33.33 |
| Cat | 0.66 | 33.33 |

**Explanation:**
Dog queries quality is ((5 / 1) + (5 / 2) + (1 / 200)) / 3 = 2.50
Dog queries poor_ query_percentage is (1 / 3) * 100 = 33.33

Cat queries quality equals ((2 / 5) + (3 / 3) + (4 / 7)) / 3 = 0.66
Cat queries poor_ query_percentage is (1 / 3) * 100 = 33.33

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
# Write your MySQL query statement below
SELECT
  query_name,
  ROUND(AVG(rating * 1.0 / position), 2) AS quality,
  ROUND(
    SUM(CASE WHEN rating < 3 THEN 1 ELSE 0 END) * 100.0 / COUNT(*),
    2
  ) AS poor_query_percentage
FROM Queries
GROUP BY query_name;
```

# 1942 Primary Department for Each Employee (link)

## Description

Table: `Employee`

```
+---------------+---------+
| Column Name   |  Type   |
+---------------+---------+
| employee_id   | int     |
| department_id | int     |
| primary_flag  | varchar |
+---------------+---------+
(employee_id, department_id) is the primary key (combination of columns with unique values) for this table.
employee_id is the id of the employee.
department_id is the id of the department to which the employee belongs.
primary_flag is an ENUM (category) of type ('Y', 'N'). If the flag is 'Y', the department is the primary department for the employee
```

Employees can belong to multiple departments. When the employee joins other departments, they need to decide which department is their primary department. Note that when an employee belongs to only one department, their primary column is `'N'`.

Write a solution to report all the employees with their primary department. For employees who belong to one department, report their only department.

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

The result format is in the following example.

**Example 1:**

```
Input:
Employee table:
+-------------+---------------+--------------+
```

```
| employee_id | department_id | primary_flag |
+-------------+---------------+--------------+
| 1           | 1             | N            |
| 2           | 1             | Y            |
| 2           | 2             | N            |
| 3           | 3             | N            |
| 4           | 2             | N            |
| 4           | 3             | Y            |
| 4           | 4             | N            |
+-------------+---------------+--------------+
```

**Output:**

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

**Explanation:**

- The Primary department for employee 1 is 1.
- The Primary department for employee 2 is 1.
- The Primary department for employee 3 is 3.
- The Primary department for employee 4 is 3.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
# Write your MySQL query statement below
SELECT employee_id, department_id
FROM Employee
WHERE primary_flag = 'Y'

UNION ALL

SELECT employee_id, department_id
FROM Employee
GROUP BY employee_id
HAVING COUNT(*) = 1;
```

# 1882 The Number of Employees Which Report to Each Employee (link)

## Description

Table: `Employees`

```
+-------------+----------+
| Column Name | Type     |
+-------------+----------+
| employee_id | int      |
| name        | varchar  |
| reports_to  | int      |
| age         | int      |
+-------------+----------+
employee_id is the column with unique values for this table.
This table contains information about the employees and the id of the manager they report to. Some employees do not report to anyone
```

For this problem, we will consider a **manager** an employee who has at least 1 other employee reporting to them.

Write a solution to report the ids and the names of all **managers**, the number of employees who report **directly** to them, and the average age of the reports rounded to the nearest integer.

Return the result table ordered by `employee_id`.

The result format is in the following example.

**Example 1:**

```
Input:
Employees table:
+-------------+---------+------------+-----+
| employee_id | name    | reports_to | age |
+-------------+---------+------------+-----+
```

```
| 9           | Hercy   | null       | 43  |
| 6           | Alice   | 9          | 41  |
| 4           | Bob     | 9          | 36  |
| 2           | Winston | null       | 37  |
+-------------+---------+------------+-----+
Output:
+-------------+-------+---------------+-------------+
| employee_id | name  | reports_count | average_age |
+-------------+-------+---------------+-------------+
| 9           | Hercy | 2             | 39          |
+-------------+-------+---------------+-------------+
Explanation: Hercy has 2 people report directly to him, Alice and Bob. Their average age is (41+36)/2 = 38.5, which is 39 after rou
```

### Example 2:

```
Input:
Employees table:
+-------------+---------+------------+-----+
| employee_id | name    | reports_to | age |
|-------------|---------|------------|-----|
| 1           | Michael | null       | 45  |
| 2           | Alice   | 1          | 38  |
| 3           | Bob     | 1          | 42  |
| 4           | Charlie | 2          | 34  |
| 5           | David   | 2          | 40  |
| 6           | Eve     | 3          | 37  |
| 7           | Frank   | null       | 50  |
| 8           | Grace   | null       | 48  |
+-------------+---------+------------+-----+
Output:
+-------------+---------+---------------+-------------+
| employee_id | name    | reports_count | average_age |
| ----------- | ------- | ------------- | ----------- |
| 1           | Michael | 2             | 40          |
| 2           | Alice   | 2             | 37          |
| 3           | Bob     | 1             | 37          |
+-------------+---------+---------------+-------------+
```

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
# Write your MySQL query statement below
SELECT
  m.employee_id,
  m.name,
  COUNT(*) AS reports_count,
  ROUND(AVG(e.age)) AS average_age
FROM Employees e
JOIN Employees m
  ON e.reports_to = m.employee_id
GROUP BY m.employee_id, m.name
ORDER BY m.employee_id;
```

# 1773 Percentage of Users Attended a Contest (link)

## Description

Table: `Users`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| user_id     | int     |
| user_name   | varchar |
+-------------+---------+
user_id is the primary key (column with unique values) for this table.
Each row of this table contains the name and the id of a user.
```

Table: `Register`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| contest_id  | int     |
| user_id     | int     |
+-------------+---------+
(contest_id, user_id) is the primary key (combination of columns with unique values) for this table.
Each row of this table contains the id of a user and the contest they registered into.
```

Write a solution to find the percentage of the users registered in each contest rounded to **two decimals**.

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

The result format is in the following example.

### Example 1:

**Input:**
Users table:

| user_id | user_name |
|---|---|
| 6 | Alice |
| 2 | Bob |
| 7 | Alex |

Register table:

| contest_id | user_id |
|---|---|
| 215 | 6 |
| 209 | 2 |
| 208 | 2 |
| 210 | 6 |
| 208 | 6 |
| 209 | 7 |
| 209 | 6 |
| 215 | 7 |
| 208 | 7 |
| 210 | 2 |
| 207 | 2 |
| 210 | 7 |

**Output:**

| contest_id | percentage |
|---|---|
| 208 | 100.0 |
| 209 | 100.0 |
| 210 | 100.0 |
| 215 | 66.67 |
| 207 | 33.33 |

**Explanation:**
All the users registered in contests 208, 209, and 210. The percentage is 100% and we sort them in the answer table by contest_id in
Alice and Alex registered in contest 215 and the percentage is ((2/3) * 100) = 66.67%
Bob registered in contest 207 and the percentage is ((1/3) * 100) = 33.33%

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT contest_id, percentage
FROM (
    SELECT
        r.contest_id,
        ROUND(COUNT(r.user_id) * 100.0 / (SELECT COUNT(*) FROM Users), 2) AS percentage
    FROM Register r
    GROUP BY r.contest_id
) t
ORDER BY percentage DESC, contest_id ASC;
```

# 1161 Project Employees I (link)

## Description

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.
Each row of this table indicates that the employee with employee_id is working on the project with project_id.
```

Table: `Employee`

```
+------------------+---------+
| Column Name      | Type    |
+------------------+---------+
| employee_id      | int     |
| name             | varchar |
| experience_years | int     |
+------------------+---------+
employee_id is the primary key of this table. It's guaranteed that experience_years is not NULL.
Each row of this table contains information about one employee.
```

Write an SQL query that reports the **average** experience years of all the employees for each project, **rounded to 2 digits**.

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

The query result format is in the following example.

### Example 1:

**Input:**
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 | 1 |
| 4 | Doe | 2 |

**Output:**

| project_id | average_years |
|---|---|
| 1 | 2.00 |
| 2 | 2.50 |

**Explanation:** The average experience years for the first project is (3 + 2 + 1) / 3 = 2.00 and for the second project is (3 + 2) / 2

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
# Write your MySQL query statement below
select p.project_id, ROUND (SUM(e.experience_years)/COUNT(e.experience_years), 2) as average_years
from Project p
left join Employee e
on p.employee_id = e.employee_id
group by p.project_id;
```

# 1390 Average Selling Price (link)

## Description

Table: `Prices`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| product_id    | int     |
| start_date    | date    |
| end_date      | date    |
| price         | int     |
+---------------+---------+
(product_id, start_date, end_date) is the primary key (combination of columns with unique values) for this table.
Each row of this table indicates the price of the product_id in the period from start_date to end_date.
For each product_id there will be no two overlapping periods. That means there will be no two intersecting periods for the same proc
```

Table: `UnitsSold`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| product_id    | int     |
| purchase_date | date    |
| units         | int     |
+---------------+---------+
This table may contain duplicate rows.
Each row of this table indicates the date, units, and product_id of each product sold.
```

Write a solution to find the average selling price for each product. `average_price` should be **rounded to 2 decimal places**. If a product does not have any sold units, its average selling price is assumed to be 0.

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

The result format is in the following example.

### Example 1:

**Input:**
Prices table:

| product_id | start_date | end_date | price |
|---|---|---|---|
| 1 | 2019-02-17 | 2019-02-28 | 5 |
| 1 | 2019-03-01 | 2019-03-22 | 20 |
| 2 | 2019-02-01 | 2019-02-20 | 15 |
| 2 | 2019-02-21 | 2019-03-31 | 30 |

UnitsSold table:

| product_id | purchase_date | units |
|---|---|---|
| 1 | 2019-02-25 | 100 |
| 1 | 2019-03-01 | 15 |
| 2 | 2019-02-10 | 200 |
| 2 | 2019-03-22 | 30 |

**Output:**

| product_id | average_price |
|---|---|
| 1 | 6.96 |
| 2 | 16.96 |

**Explanation:**
Average selling price = Total Price of Product / Number of products sold.
Average selling price for product 1 = ((100 * 5) + (15 * 20)) / 115 = 6.96
Average selling price for product 2 = ((200 * 15) + (30 * 30)) / 230 = 16.96

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
# Write your MySQL query statement below
select p.product_id, IFNULL(ROUND(SUM(p.price * u.units)/SUM(u.units), 2), 0) as average_price
from Prices p
left  join UnitsSold u
on  p.product_id = u.product_id
AND u.purchase_date BETWEEN p.start_date AND p.end_date
group by p.product_id;
```

# 620 Not Boring Movies (link)

## Description

Table: `Cinema`

```
+----------------+----------+
| Column Name    | Type     |
+----------------+----------+
| id             | int      |
| movie          | varchar  |
| description    | varchar  |
| rating         | float    |
+----------------+----------+
id is the primary key (column with unique values) for this table.
Each row contains information about the name of a movie, its genre, and its rating.
rating is a 2 decimal places float in the range [0, 10]
```

Write a solution to report the movies with an odd-numbered ID and a description that is not `"boring"`.

Return the result table ordered by `rating` **in descending order**.

The result format is in the following example.

**Example 1:**

```
Input:
Cinema table:
+----+------------+-------------+--------+
| id | movie      | description | rating |
+----+------------+-------------+--------+
| 1  | War        | great 3D    | 8.9    |
| 2  | Science    | fiction     | 8.5    |
| 3  | irish      | boring      | 6.2    |
| 4  | Ice song   | Fantacy     | 8.6    |
```

```
| 5  | House card | Interesting | 9.1    |
+----+------------+-------------+--------+
```
**Output:**
```
+----+------------+-------------+--------+
| id | movie      | description | rating |
+----+------------+-------------+--------+
| 5  | House card | Interesting | 9.1    |
| 1  | War        | great 3D    | 8.9    |
+----+------------+-------------+--------+
```
**Explanation:**
We have three movies with odd-numbered IDs: 1, 3, and 5. The movie with ID = 3 is boring so we do not include it in the answer.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
# Write your MySQL query statement below
select * from Cinema
where (id%2 != 0) and (description != "boring")
order by rating desc;
```

# 2087 Confirmation Rate (link)

## Description

Table: `Signups`

```
+----------------+----------+
| Column Name    | Type     |
+----------------+----------+
| user_id        | int      |
| time_stamp     | datetime |
+----------------+----------+
user_id is the column of unique values for this table.
Each row contains information about the signup time for the user with ID user_id.
```

Table: `Confirmations`

```
+----------------+----------+
| Column Name    | Type     |
+----------------+----------+
| user_id        | int      |
| time_stamp     | datetime |
| action         | ENUM     |
+----------------+----------+
(user_id, time_stamp) is the primary key (combination of columns with unique values) for this table.
user_id is a foreign key (reference column) to the Signups table.
action is an ENUM (category) of the type ('confirmed', 'timeout')
Each row of this table indicates that the user with ID user_id requested a confirmation message at time_stamp and that confirmation
```

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

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

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

The result format is in the following example.

### Example 1:

**Input:**
Signups table:

| user_id | time_stamp |
|---|---|
| 3 | 2020-03-21 10:16:13 |
| 7 | 2020-01-04 13:57:59 |
| 2 | 2020-07-29 23:09:44 |
| 6 | 2020-12-09 10:39:37 |

Confirmations table:

| user_id | time_stamp | action |
|---|---|---|
| 3 | 2021-01-06 03:30:46 | timeout |
| 3 | 2021-07-14 14:00:00 | timeout |
| 7 | 2021-06-12 11:57:29 | confirmed |
| 7 | 2021-06-13 12:58:28 | confirmed |
| 7 | 2021-06-14 13:59:27 | confirmed |
| 2 | 2021-01-22 00:00:00 | confirmed |
| 2 | 2021-02-28 23:59:59 | timeout |

**Output:**

| user_id | confirmation_rate |
|---|---|
| 6 | 0.00 |
| 3 | 0.00 |
| 7 | 1.00 |
| 2 | 0.50 |

**Explanation:**
User 6 did not request any confirmation messages. The confirmation rate is 0.
User 3 made 2 requests and both timed out. The confirmation rate is 0.

```
User 7 made 3 requests and all were confirmed. The confirmation rate is 1.
User 2 made 2 requests where one was confirmed and the other timed out. The confirmation rate is 1 / 2 = 0.5.
```

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT
    s.user_id,
    ROUND(
        IFNULL(SUM(c.action = 'confirmed') / COUNT(c.action), 0),
        2
    ) AS confirmation_rate
FROM Signups s
LEFT JOIN Confirmations c
    ON s.user_id = c.user_id
GROUP BY s.user_id;
```

# 570 Managers with at Least 5 Direct Reports (link)

## Description

Table: `Employee`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| name        | varchar |
| department  | varchar |
| managerId   | int     |
+-------------+---------+
id is the primary key (column with unique values) for this table.
Each row of this table indicates the name of an employee, their department, and the id of their manager.
If managerId is null, then the employee does not have a manager.
No employee will be the manager of themself.
```

Write a solution to find managers with at least **five direct reports**.

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

The result format is in the following example.

**Example 1:**

```
Input:
Employee table:
+-----+-------+------------+-----------+
| 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       |
+-----+-------+------------+-----------+
```

**Output:**

```
+------+
| name |
+------+
| John |
+------+
```

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT m.name
FROM Employee e
JOIN Employee m
  ON e.managerId = m.id
GROUP BY m.id, m.name
HAVING COUNT(e.id) >= 5;
```

# 1415 Students and Examinations (link)

## Description

Table: `Students`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| student_id    | int     |
| student_name  | varchar |
+---------------+---------+
student_id is the primary key (column with unique values) for this table.
Each row of this table contains the ID and the name of one student in the school.
```

Table: `Subjects`

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| subject_name | varchar |
+--------------+---------+
subject_name is the primary key (column with unique values) for this table.
Each row of this table contains the name of one subject in the school.
```

Table: `Examinations`

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| student_id   | int     |
| subject_name | varchar |
+--------------+---------+
There is no primary key (column with unique values) for this table. It may contain duplicates.
```

```
Each student from the Students table takes every course from the Subjects table.
Each row of this table indicates that a student with ID student_id attended the exam of subject_name.
```

Write a solution to find the number of times each student attended each exam.

Return the result table ordered by `student_id` and `subject_name`.

The result format is in the following example.

**Example 1:**

```
Input:
Students table:
+------------+--------------+
| student_id | student_name |
+------------+--------------+
| 1          | Alice        |
| 2          | Bob          |
| 13         | John         |
| 6          | Alex         |
+------------+--------------+
Subjects table:
+--------------+
| subject_name |
+--------------+
| Math         |
| Physics      |
| Programming  |
+--------------+
Examinations table:
+------------+--------------+
| student_id | subject_name |
+------------+--------------+
| 1          | Math         |
| 1          | Physics      |
| 1          | Programming  |
| 2          | Programming  |
| 1          | Physics      |
| 1          | Math         |
| 13         | Math         |
```

```
| 13         | Programming  |
| 13         | Physics      |
| 2          | Math         |
| 1          | Math         |
+------------+--------------+
```

**Output:**

```
+------------+--------------+--------------+----------------+
| student_id | student_name | subject_name | attended_exams |
+------------+--------------+--------------+----------------+
| 1          | Alice        | Math         | 3              |
| 1          | Alice        | Physics      | 2              |
| 1          | Alice        | Programming  | 1              |
| 2          | Bob          | Math         | 1              |
| 2          | Bob          | Physics      | 0              |
| 2          | Bob          | Programming  | 1              |
| 6          | Alex         | Math         | 0              |
| 6          | Alex         | Physics      | 0              |
| 6          | Alex         | Programming  | 0              |
| 13         | John         | Math         | 1              |
| 13         | John         | Physics      | 1              |
| 13         | John         | Programming  | 1              |
+------------+--------------+--------------+----------------+
```

**Explanation:**

The result table should contain all students and all subjects.
Alice attended the Math exam 3 times, the Physics exam 2 times, and the Programming exam 1 time.
Bob attended the Math exam 1 time, the Programming exam 1 time, and did not attend the Physics exam.
Alex did not attend any exams.
John attended the Math exam 1 time, the Physics exam 1 time, and the Programming exam 1 time.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
# Write your MySQL query statement below
 select S1.student_id, S1.student_name, S2. subject_name, COUNT(E.subject_name)  as attended_exams
 from Students S1
 cross join Subjects S2
 left join Examinations E
 on s1.student_id = E.student_id
 and s2.subject_name = E.subject_name
 group by s1.student_id, s1.student_name, s2.subject_name
 order by s1.student_id, s2.subject_name;
```

# 577 Employee Bonus (link)

## Description

Table: `Employee`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| empId       | int     |
| name        | varchar |
| supervisor  | int     |
| salary      | int     |
+-------------+---------+
empId is the column with unique values for this table.
Each row of this table indicates the name and the ID of an employee in addition to their salary and the id of their manager.
```

Table: `Bonus`

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| empId       | int  |
| bonus       | int  |
+-------------+------+
empId is the column of unique values for this table.
empId is a foreign key (reference column) to empId from the Employee table.
Each row of this table contains the id of an employee and their respective bonus.
```

Write a solution to report the name and bonus amount of each employee who satisfies either of the following:

- The employee has a bonus **less than** `1000`.
- The employee did not get any bonus.

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

The result format is in the following example.

### Example 1:

**Input:**

Employee table:

| empId | name | supervisor | salary |
|---|---|---|---|
| 3 | Brad | null | 4000 |
| 1 | John | 3 | 1000 |
| 2 | Dan | 3 | 2000 |
| 4 | Thomas | 3 | 4000 |

Bonus table:

| empId | bonus |
|---|---|
| 2 | 500 |
| 4 | 2000 |

**Output:**

| name | bonus |
|---|---|
| Brad | null |
| John | null |
| Dan | 500 |

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
select e.name, b.bonus
from Employee e
left join Bonus b
on e.empId = b.empId
where b.bonus < 1000 or b.bonus IS NULL;
```

# 1801 Average Time of Process per Machine (link)

## Description

Table: `Activity`

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| machine_id     | int     |
| process_id     | int     |
| activity_type  | enum    |
| timestamp      | float   |
+----------------+---------+
The table shows the user activities for a factory website.
(machine_id, process_id, activity_type) is the primary key (combination of columns with unique values) of this table.
machine_id is the ID of a machine.
process_id is the ID of a process running on the machine with ID machine_id.
activity_type is an ENUM (category) of type ('start', 'end').
timestamp is a float representing the current time in seconds.
'start' means the machine starts the process at the given timestamp and 'end' means the machine ends the process at the given timest
The 'start' timestamp will always be before the 'end' timestamp for every (machine_id, process_id) pair.
It is guaranteed that each (machine_id, process_id) pair has a 'start' and 'end' timestamp.
```

There is a factory website that has several machines each running the **same number of processes**. Write a solution to find the **average time** each machine takes to complete a process.

The time to complete a process is the `'end' timestamp` minus the `'start' timestamp`. The average time is calculated by the total time to complete every process on the machine divided by the number of processes that were run.

The resulting table should have the `machine_id` along with the **average time** as `processing_time`, which should be **rounded to 3 decimal places**.

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

The result format is in the following example.

### Example 1:

**Input:**
Activity table:

| machine_id | process_id | activity_type | timestamp |
|---|---|---|---|
| 0 | 0 | start | 0.712 |
| 0 | 0 | end | 1.520 |
| 0 | 1 | start | 3.140 |
| 0 | 1 | end | 4.120 |
| 1 | 0 | start | 0.550 |
| 1 | 0 | end | 1.550 |
| 1 | 1 | start | 0.430 |
| 1 | 1 | end | 1.420 |
| 2 | 0 | start | 4.100 |
| 2 | 0 | end | 4.512 |
| 2 | 1 | start | 2.500 |
| 2 | 1 | end | 5.000 |

**Output:**

| machine_id | processing_time |
|---|---|
| 0 | 0.894 |
| 1 | 0.995 |
| 2 | 1.456 |

**Explanation:**
There are 3 machines running 2 processes each.
Machine 0's average time is ((1.520 - 0.712) + (4.120 - 3.140)) / 2 = 0.894
Machine 1's average time is ((1.550 - 0.550) + (1.420 - 0.430)) / 2 = 0.995
Machine 2's average time is ((4.512 - 4.100) + (5.000 - 2.500)) / 2 = 1.456

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
select a1.machine_id, ROUND (AVG(a1.timestamp - a2.timestamp), 3) as processing_time
from Activity a1
join Activity a2
ON a1.machine_id = a2.machine_id
and a1.process_id = a2.process_id
and a1.activity_type = 'end'
and a2.activity_type = 'start'
group by a1.machine_id;
```

# 197 Rising Temperature (link)

## Description

Table: `Weather`

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

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

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

The result format is in the following example.

**Example 1:**

```
Input:
Weather table:
+----+------------+-------------+
| id | recordDate | temperature |
+----+------------+-------------+
| 1  | 2015-01-01 | 10          |
| 2  | 2015-01-02 | 25          |
| 3  | 2015-01-03 | 20          |
| 4  | 2015-01-04 | 30          |
+----+------------+-------------+
```

**Output:**
```
+----+
| id |
+----+
| 2  |
| 4  |
+----+
```
**Explanation:**
In 2015-01-02, the temperature was higher than the previous day (10 -> 25).
In 2015-01-04, the temperature was higher than the previous day (20 -> 30).

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT w1.id
FROM Weather w1
JOIN Weather w2
  ON w2.recordDate = DATE_SUB(w1.recordDate, INTERVAL 1 DAY)
WHERE w1.temperature > w2.temperature;
```

# 1509 Replace Employee ID With The Unique Identifier (link)

## Description

Table: `Employees`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| name          | varchar |
+---------------+---------+
id is the primary key (column with unique values) for this table.
Each row of this table contains the id and the name of an employee in a company.
```

Table: `EmployeeUNI`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| unique_id     | int     |
+---------------+---------+
(id, unique_id) is the primary key (combination of columns with unique values) for this table.
Each row of this table contains the id and the corresponding unique id of an employee in the company.
```

Write a solution to show the **unique ID** of each user, If a user does not have a unique ID replace just show `null`.

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

The result format is in the following example.

### Example 1:

**Input:**
Employees table:

| id | name |
|---|---|
| 1 | Alice |
| 7 | Bob |
| 11 | Meir |
| 90 | Winston |
| 3 | Jonathan |

EmployeeUNI table:

| id | unique_id |
|---|---|
| 3 | 1 |
| 11 | 2 |
| 90 | 3 |

**Output:**

| unique_id | name |
|---|---|
| null | Alice |
| null | Bob |
| 2 | Meir |
| 3 | Winston |
| 1 | Jonathan |

**Explanation:**
Alice and Bob do not have a unique ID, We will show null instead.
The unique ID of Meir is 2.
The unique ID of Winston is 3.
The unique ID of Jonathan is 1.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
# Write your MySQL query statement below
select EmployeeUNI.unique_id, Employees.name
from Employees
LEFT JOIN EmployeeUNI on Employees.id = EmployeeUNI.id;
```

# 1724 Customer Who Visited but Did Not Make Any Transactions (link)

## Description

Table: `Visits`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| visit_id    | int     |
| customer_id | int     |
+-------------+---------+
visit_id is the column with unique values for this table.
This table contains information about the customers who visited the mall.
```

Table: `Transactions`

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| transaction_id | int     |
| visit_id       | int     |
| amount         | int     |
+----------------+---------+
transaction_id is column with unique values for this table.
This table contains information about the transactions made during the visit_id.
```

Write a solution to find the IDs of the users who visited without making any transactions and the number of times they made these types of visits.

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

The result format is in the following example.

### Example 1:

**Input:**
Visits

| visit_id | customer_id |
|---|---|
| 1 | 23 |
| 2 | 9 |
| 4 | 30 |
| 5 | 54 |
| 6 | 96 |
| 7 | 54 |
| 8 | 54 |

Transactions

| transaction_id | visit_id | amount |
|---|---|---|
| 2 | 5 | 310 |
| 3 | 5 | 300 |
| 9 | 5 | 200 |
| 12 | 1 | 910 |
| 13 | 2 | 970 |

**Output:**

| customer_id | count_no_trans |
|---|---|
| 54 | 2 |
| 30 | 1 |
| 96 | 1 |

**Explanation:**
Customer with id = 23 visited the mall once and made one transaction during the visit with id = 12.
Customer with id = 9 visited the mall once and made one transaction during the visit with id = 13.
Customer with id = 30 visited the mall once and did not make any transactions.
Customer with id = 54 visited the mall three times. During 2 visits they did not make any transactions, and during one visit they ma
Customer with id = 96 visited the mall once and did not make any transactions.
As we can see, users with IDs 30 and 96 visited the mall one time without making any transactions. Also, user 54 visited the mall tw

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT
    v.customer_id,
    COUNT(*) AS count_no_trans
FROM Visits v
LEFT JOIN Transactions t
    ON v.visit_id = t.visit_id
WHERE t.visit_id IS NULL
GROUP BY v.customer_id;
```

# 1153 Product Sales Analysis I (link)

## Description

Table: `Sales`

```
+-------------+-------+
| Column Name | Type  |
+-------------+-------+
| sale_id     | int   |
| product_id  | int   |
| year        | int   |
| quantity    | int   |
| price       | int   |
+-------------+-------+
(sale_id, year) is the primary key (combination of columns with unique values) of this table.
product_id is a foreign key (reference column) to Product table.
Each row of this table shows a sale on the product product_id in a certain year.
Note that the price is per unit.
```

Table: `Product`

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| product_id   | int     |
| product_name | varchar |
+--------------+---------+
product_id is the primary key (column with unique values) of this table.
Each row of this table indicates the product name of each product.
```

Write a solution to report the `product_name`, `year`, and `price` for each `sale_id` in the `Sales` table.

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

The result format is in the following example.

### Example 1:

**Input:**
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 |

**Output:**

| product_name | year | price |
|---|---|---|
| Nokia | 2008 | 5000 |
| Nokia | 2009 | 5000 |
| Apple | 2011 | 9000 |

**Explanation:**
From sale_id = 1, we can conclude that Nokia was sold for 5000 in the year 2008.
From sale_id = 2, we can conclude that Nokia was sold for 5000 in the year 2009.
From sale_id = 7, we can conclude that Apple was sold for 9000 in the year 2011.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
# Write your MySQL query statement below
select product_name, year, price
from Sales
left join product
on sales.product_id = product.product_id;
```

# 1827 Invalid Tweets (link)

## Description

Table: `Tweets`

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| tweet_id       | int     |
| content        | varchar |
+----------------+---------+
tweet_id is the primary key (column with unique values) for this table.
content consists of alphanumeric characters, '!', or ' ' and no other special characters.
This table contains all the tweets in a social media app.
```

Write a solution to find the IDs of the invalid tweets. The tweet is invalid if the number of characters used in the content of the tweet is **strictly greater** than `15`.

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

The result format is in the following example.

**Example 1:**

```
Input:
Tweets table:
+----------+----------------------------------+
| tweet_id | content                          |
+----------+----------------------------------+
| 1        | Let us Code                      |
| 2        | More than fifteen chars are here! |
+----------+----------------------------------+
Output:
+----------+
```

```
| tweet_id |
+----------+
| 2        |
+----------+
```

**Explanation:**

```
Tweet 1 has length = 11. It is a valid tweet.
Tweet 2 has length = 33. It is an invalid tweet.
```

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
# Write your MySQL query statement below
SELECT tweet_id from Tweets
WHERE LENGTH(content) > 15;
```

# 1258 Article Views I (link)

## Description

Table: `Views`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| article_id    | int     |
| author_id     | int     |
| viewer_id     | int     |
| view_date     | date    |
+---------------+---------+
There is no primary key (column with unique values) for this table, the table 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 a solution to find all the authors that viewed at least one of their own articles.

Return the result table sorted by `id` in ascending order.

The result format is in the following example.

**Example 1:**

```
Input:
Views table:
+------------+-----------+-----------+------------+
| article_id | author_id | viewer_id | view_date  |
+------------+-----------+-----------+------------+
| 1          | 3         | 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 |
+-------------+-------------+-------------+-------------+
```

**Output:**

```
+------+
| id   |
+------+
| 4    |
| 7    |
+------+
```

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
# Write your MySQL query statement below
SELECT DISTINCT author_id as id  from Views
WHERE author_id = viewer_id
ORDER BY id ASC;
```

# 595 Big Countries (link)

## Description

Table: `World`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| name        | varchar |
| continent   | varchar |
| area        | int     |
| population  | int     |
| gdp         | bigint  |
+-------------+---------+
name is the primary key (column with unique values) for this table.
Each row of this table gives information about the name of a country, the continent to which it belongs, its area, the population, a
```

A country is **big** if:

- it has an area of at least three million (i.e., `3000000 km`$^2$), or
- it has a population of at least twenty-five million (i.e., `25000000`).

Write a solution to find the name, population, and area of the **big countries**.

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

The result format is in the following example.

**Example 1:**

```
Input:
World table:
```

```
+-------------+-----------+---------+------------+--------------+
| name        | continent | area    | population | gdp          |
+-------------+-----------+---------+------------+--------------+
| Afghanistan | Asia      | 652230  | 25500100   | 20343000000  |
| Albania     | Europe    | 28748   | 2831741    | 12960000000  |
| Algeria     | Africa    | 2381741 | 37100000   | 188681000000 |
| Andorra     | Europe    | 468     | 78115      | 3712000000   |
| Angola      | Africa    | 1246700 | 20609294   | 100990000000 |
+-------------+-----------+---------+------------+--------------+
```

**Output:**

```
+-------------+------------+---------+
| name        | population | area    |
+-------------+------------+---------+
| Afghanistan | 25500100   | 652230  |
| Algeria     | 37100000   | 2381741 |
+-------------+------------+---------+
```

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT name, population, area from World
where area >= 3000000 or population >= 25000000;
```

# [584 Find Customer Referee (link)]

## Description

Table: `Customer`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| name        | varchar |
| referee_id  | int     |
+-------------+---------+
In SQL, id is the primary key column for this table.
Each row of this table indicates the id of a customer, their name, and the id of the customer who referred them.
```

Find the names of the customer that are either:

1. **referred by** any customer with `id != 2`.
2. **not referred by** any customer.

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

The result format is in the following example.

**Example 1:**

```
Input:
Customer table:
+----+------+------------+
| id | name | referee_id |
+----+------+------------+
| 1  | Will | null       |
| 2  | Jane | null       |
| 3  | Alex | 2          |
```

```
| 4  | Bill | null       |
| 5  | Zack | 1          |
| 6  | Mark | 2          |
+----+------+------------+
```

**Output:**

```
+------+
| name |
+------+
| Will |
| Jane |
| Bill |
| Zack |
+------+
```

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
# Write your MySQL query statement below
select name from Customer
where referee_id != 2 or referee_id IS NULL;
```

# 1908 Recyclable and Low Fat Products (link)

## Description

Table: `Products`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| product_id  | int     |
| low_fats    | enum    |
| recyclable  | enum    |
+-------------+---------+
product_id is the primary key (column with unique values) for this table.
low_fats is an ENUM (category) of type ('Y', 'N') where 'Y' means this product is low fat and 'N' means it is not.
recyclable is an ENUM (category) of types ('Y', 'N') where 'Y' means this product is recyclable and 'N' means it is not.
```

Write a solution to find the ids of products that are both low fat and recyclable.

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

The result format is in the following example.

**Example 1:**

```
Input:
Products table:
+-------------+----------+------------+
| product_id  | low_fats | recyclable |
+-------------+----------+------------+
| 0           | Y        | N          |
| 1           | Y        | Y          |
| 2           | N        | Y          |
| 3           | Y        | Y          |
| 4           | N        | N          |
```

```
+-------------+----------+------------+
```
**Output:**
```
+-------------+
| product_id  |
+-------------+
| 1           |
| 3           |
+-------------+
```
**Explanation:** Only products 1 and 3 are both low fat and recyclable.

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT product_id from Products
WHERE low_fats = 'Y' AND recyclable = 'Y';
```

# 178 Rank Scores (link)

## Description

Table: `Scores`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| score       | decimal |
+-------------+---------+
id is the primary key (column with unique values) for this table.
Each row of this table contains the score of a game. Score is a floating point value with two decimal places.
```

Write a solution to find the rank of the scores. The ranking should be calculated according to the following rules:

- The scores should be ranked from the highest to the lowest.
- If there is a tie between two scores, both should have the same ranking.
- After a tie, the next ranking number should be the next consecutive integer value. In other words, there should be no holes between ranks.

Return the result table ordered by `score` in descending order.

The result format is in the following example.

**Example 1:**

```
Input:
Scores table:
+----+-------+
| id | score |
+----+-------+
| 1  | 3.50  |
```

```
| 2  | 3.65  |
| 3  | 4.00  |
| 4  | 3.85  |
| 5  | 4.00  |
| 6  | 3.65  |
+----+-------+
```

**Output:**

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

(scroll down for solution)

## Solution

*Language: mysql*

**Status: Accepted**

```
SELECT
    score,
    DENSE_RANK() OVER (ORDER BY score DESC) AS 'rank'
FROM Scores;
```