**SQL HACKER RANK**

* **BASIC SELECT**

1. **Revising the Select Query I**

Query all columns for all American cities in the **CITY** table with populations larger than 100000. The **CountryCode** for America is USA.

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



**Ans:**

**select \* from city where population>100000 and countrycode='USA'**

1. **Revising the Select Query II**

Query the **NAME** field for all American cities in the **CITY** table with populations larger than 120000. The CountryCode for America is USA.

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


**Ans:**

**select name from city where population >120000 and countrycode='USA'**

1. **Select ALL**

Query all columns (attributes) for every row in the **CITY** table.

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


**Ans:**

**select \* from city ;**

1. **Select By ID**

Query all columns for a city in **CITY** with the ID 1661.

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


**Ans:**

**select \* from city where ID=1661 ;**

1. **Japanese Cities Attributes**

Query all attributes of every Japanese city in the **CITY** table. The **COUNTRYCODE** for Japan is JPN.

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


**Ans:**

**select \* from city where COUNTRYCODE = 'JPN' ;**

1. **Japanese Cities Names**

Query the names of all the Japanese cities in the **CITY** table. The **COUNTRYCODE** for Japan is JPN.  
The **CITY** table is described as follows:  


**Ans:**

**select name from city where countrycode='JPN';**

1. **Weather Observation Station 1**

Query a list of **CITY** and **STATE** from the **STATION** table.  
The **STATION** table is described as follows:

  
where **LAT\_N** is the northern latitude and **LONG\_W** is the western longitude.

**Ans: select city,state from station ;**

1. **Weather Observation Station 3**

Query a list of **CITY** names from **STATION** for cities that have an even **ID** number. Print the results in any order, but exclude duplicates from the answer.  
The **STATION** table is described as follows:



where **LAT\_N** is the northern latitude and **LONG\_W** is the western longitude.

**Ans:**

**select distinct city from station where ID%2 = 0 ;**

1. **Weather Observation Station 4**

Find the difference between the total number of **CITY** entries in the table and the number of distinct **CITY** entries in the table.  
The **STATION** table is described as follows:



where **LAT\_N** is the northern latitude and **LONG\_W** is the western longitude.

For example, if there are three records in the table with **CITY** values 'New York', 'New York', 'Bengalaru', there are 2 different city names: 'New York' and 'Bengalaru'. The query returns 1, because

Total no. of records – number of unique city names = 3-2 =1

**Ans:**

**select count(city) - count(distinct city) from station ;**

1. **Weather Observation Station 5**

Query the two cities in **STATION** with the shortest and longest *CITY* names, as well as their respective lengths (i.e.: number of characters in the name). If there is more than one smallest or largest city, choose the one that comes first when ordered alphabetically.  
The **STATION** table is described as follows:



where **LAT\_N** is the northern latitude and **LONG\_W** is the western longitude.

**Sample Input**

For example, **CITY** has four entries: **DEF, ABC, PQRS** and **WXY**.

**Sample Output**

ABC 3

PQRS 4

**Explanation**

When ordered alphabetically, the **CITY** names are listed as **ABC, DEF, PQRS,** and **WXY**, with lengths 3,3,4 and 3 . The longest name is **PQRS**, but there are  3 options for shortest named city. Choose **ABC**, because it comes first alphabetically.

**Note**  
You can write two separate queries to get the desired output. It need not be a single query.

**Ans:**

**select CITY, length(CITY) from STATION order by length(CITY), CITY limit 1;**

**select CITY, length(CITY) from STATION order by length(CITY) desc, CITY limit 1;**

1. **Weather Observation Station 6**

Query the list of CITY names starting with vowels (i.e., a, e, i, o, or u) from **STATION**. Your result cannot contain duplicates.

**Input Format**

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



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Ans:**

**SELECT DISTINCT(CITY) FROM STATION WHERE CITY LIKE 'A%' OR CITY LIKE 'E%' OR CITY LIKE 'I%' OR CITY LIKE 'O%' OR CITY LIKE 'U%' ORDER BY CITY ASC ;**

1. **Weather Observation Station 7**

Query the list of CITY names ending with vowels (a, e, i, o, u) from **STATION**. Your result cannot contain duplicates.

**Input Format**

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



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Ans:**

**select distinct city from station where city like '%a' or city like '%e' or city like '%i' or city like '%o' or city like '%u';**

1. **Weather Observation Station 8**

Query the list of CITY names from **STATION** which have vowels (i.e., a, e, i, o, and u) as both their first and last characters. Your result cannot contain duplicates.

**Input Format**

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



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Ans:**

**select distinct city from station where city like '%a' or city like '%e' or city like '%i' or city like '%o' or city like '%u';**

1. **Weather Observation Station 9**

Query the list of CITY names from **STATION** that do not start with vowels. Your result cannot contain duplicates.

**Input Format**

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



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Ans:**

**select distinct city from station where city not like 'a%' or city not like 'e%' or city not like 'i%' or city not like 'o%' or city not like 'u%'**

1. **Weather Observation Station 10**

Query the list of CITY names from **STATION** that do not end with vowels. Your result cannot contain duplicates.

**Input Format**

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



where LAT\_N is the northern latitude and LONG\_W is the western longitude

**Ans:**

**select distinct city from station where city not like '%a' and city not like '%e' and city not like '%i' and city not like '%o' and city not like '%u'**

1. **Weather Observation Station 11**

Query the list of CITY names from **STATION** that either do not start with vowels or do not end with vowels. Your result cannot contain duplicates.

**Input Format**

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



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Ans:**

**select distinct city from station where (city not like '%a' and city not like '%e' and city not like '%i' and city not like '%o' and city not like '%u') or (city not like 'a%' and city not like 'e%' and city not like 'i%' and city not like 'o%' and city not like 'u%')**

1. **Weather Observation Station 12**

Query the list of CITY names from **STATION** that either do not start with vowels and do not end with vowels. Your result cannot contain duplicates.

**Input Format**

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



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Ans:**

**select distinct city from station where (city not like '%a' and city not like '%e' and city not like '%i' and city not like '%o' and city not like '%u') and (city not like 'a%' and city not like 'e%' and city not like 'i%' and city not like 'o%' and city not like 'u%')**

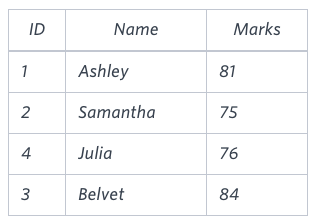
1. **Higher than 75 Marks**

Query the Name of any student in **STUDENTS** who scored higher than  75 Marks. Order your output by the last three characters of each name. If two or more students both have names ending in the same last three characters (i.e.: Bobby, Robby, etc.), secondary sort them by ascending ID.

**Input Format**

The **STUDENTS** table is described as follows:  The Name column only contains uppercase (A-Z) and lowercase (a-z) letters.

**Sample Input**



**Sample Output**

Ashley

Julia

Belvet

**Explanation**

Only Ashley, Julia, and Belvet have Marks > 75 . If you look at the last three characters of each of their names, there are no duplicates and 'ley' < 'lia' < 'vet'.

**Ans:**

**select name from students where marks >75**

**order by right(name,3) asc, ID asc;**

1. **Employee Names**

Write a query that prints a list of employee names (i.e.: the name attribute) from the **Employee** table in alphabetical order.

**Input Format**

The **Employee** table containing employee data for a company is described as follows:



where employee\_id is an employee's ID number, name is their name, months is the total number of months they've been working for the company, and salary is their monthly salary.

**Sample Input**



**Sample Output**

Angela

Bonnie

Frank

Joe

Kimberly

Lisa

Michael

Patrick

Rose

Todd

**Ans:**

**select name from employee order by name;**

1. **Employee Salaries**

Write a query that prints a list of employee names (i.e.: the name attribute) for employees in **Employee** having a salary greater than *$2000* per month who have been employees for less than *10* months. Sort your result by ascending employee\_id.

**Input Format**

The **Employee** table containing employee data for a company is described as follows:



where employee\_id is an employee's ID number, name is their name, months is the total number of months they've been working for the company, and salary is the their monthly salary.

**Sample Input**



**Sample Output**

Angela

Michael

Todd

Joe

**Explanation**

Angela has been an employee for 1 month and earns $3443 per month.  
Michael has been an employee for 6 months and earns $2017 per month.  
Todd has been an employee for 5 months and earns $3396 per month.  
Joe has been an employee for 9 months and earns $3573 per month.  
We order our output by ascending employee\_id.

**Ans:**

**select name from employee where salary > 2000 and months < 10**

**order by employee\_id asc;**

* **ADVANCE SELECT**

1. **Type of Triangle**

Write a query identifying the *type* of each record in the **TRIANGLES** table using its three side lengths. Output one of the following statements for each record in the table:

* **Equilateral**: It's a triangle with 3  sides of equal length.
* **Isosceles**: It's a triangle with  2 sides of equal length.
* **Scalene**: It's a triangle with  3 sides of differing lengths.
* **Not A Triangle**: The given values of *A*, *B*, and *C* don't form a triangle.

**Input Format**

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



Each row in the table denotes the lengths of each of a triangle's three sides.

**Sample Input**



**Sample Output**

Isosceles

Equilateral

Scalene

Not A Triangle

**Explanation**

Values in the tuple  form an Isosceles triangle, because .  
Values in the tuple  form an Equilateral triangle, because . Values in the tuple  form a Scalene triangle, because .  
Values in the tuple  cannot form a triangle because the combined value of sides  and  is not larger than that of side .

**Ans:**

**MYSQL**

**SELECT**

**CASE**

**WHEN (A + B <= C) OR (B + C <= A) OR (A + C <= B) THEN 'Not A Triangle'**

**WHEN (A = B) AND (B = C) THEN 'Equilateral'**

**WHEN ((A = B) &(A != C)) OR ((B = C) &(B != A)) OR ((A = C) &(A != B)) THEN 'Isosceles'**

**WHEN (A != B) AND (B != C) AND (A != C) THEN 'Scalene'**

**END AS Triangle\_Type**

**FROM**

**TRIANGLES;**

**MS SQL SERVER**

**SELECT CASE**

**WHEN A + B <= C OR A + C <= B OR B + C <= A THEN 'Not A Triangle'**

**WHEN A = B AND B = C THEN 'Equilateral'**

**WHEN A = B OR B = C OR A = C THEN 'Isosceles'**

**ELSE 'Scalene'**

**END**

**FROM TRIANGLES;**

1. **The PADS**

Generate the following two result sets:

1. Query an alphabetically ordered list of all names in OCCUPATIONS, immediately followed by the first letter of each profession as a parenthetical (i.e.: enclosed in parentheses). For example: AnActorName(A), ADoctorName(D), AProfessorName(P), and ASingerName(S).
2. Query the number of ocurrences of each occupation in OCCUPATIONS. Sort the occurrences in ascending order, and output them in the following format:  
   There are a total of [occupation\_count] [occupation]s.  
   where [occupation\_count] is the number of occurrences of an occupation in OCCUPATIONS and [occupation] is the lowercase occupation name. If more than one Occupation has the same [occupation\_count], they should be ordered alphabetically.

***Note:*** There will be at least two entries in the table for each type of occupation.

***Input Format***  
The OCCUPATIONS table is described as follows:

Occupation will only contain one of the following values: Doctor, Professor, Singer or Actor.

***Sample Input***  
An OCCUPATIONS table that contains the following records:

***Sample Output***  
Ashely(P)  
Christeen(P)  
Jane(A)  
Jenny(D)  
Julia(A)  
Ketty(P)  
Maria(A)  
Meera(S)  
Priya(S)  
Samantha(D)  
There are a total of 2 doctors.  
There are a total of 2 singers.  
There are a total of 3 actors.  
There are a total of 3 professors.

**Ans:**

**MYSQL**

**select concat(name,"(",left(occupation,1),")") from occupations**

**order by name asc;**

**select concat("There are a total of ",count(occupation)," ",lower(occupation), "s.") from occupations**

**group by occupation**

**order by count(occupation) asc,occupation;**

1. **Occupations**

[Pivot](https://en.wikipedia.org/wiki/Pivot_table) the Occupation column in **OCCUPATIONS** so that each Name is sorted alphabetically and displayed underneath its corresponding Occupation. The output column headers should be Doctor, Professor, Singer, and Actor, respectively.

**Note:** Print **NULL** when there are no more names corresponding to an occupation.

**Input Format**

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



Occupation will only contain one of the following values: **Doctor**, **Professor**, **Singer** or **Actor**.

**Sample Input**



**Sample Output**

Jenny Ashley Meera Jane

Samantha Christeen Priya Julia

NULL Ketty NULL Maria

**Explanation**

The first column is an alphabetically ordered list of Doctor names.  
The second column is an alphabetically ordered list of Professor names.  
The third column is an alphabetically ordered list of Singer names.  
The fourth column is an alphabetically ordered list of Actor names.  
The empty cell data for columns with less than the maximum number of names per occupation (in this case, the Professor and Actor columns) are filled with **NULL** values.

**Ans:**

**MYSQL**

**CREATE VIEW pq AS (**

**SELECT**

**CASE WHEN occupation = 'Doctor' THEN name END AS 'Doctor',**

**CASE WHEN occupation = 'Professor' THEN name END AS 'Professor',**

**CASE WHEN occupation = 'Singer' THEN name END AS 'Singer',**

**CASE WHEN occupation = 'Actor' THEN name END AS 'Actor',**

**ROW\_NUMBER() OVER (PARTITION BY occupation ORDER BY name) as cr**

**FROM occupations**

**);**

**SELECT MAX(Doctor),MAX(Professor),MAX(Singer),MAX(Actor) FROM pq**

**GROUP BY cr;**

**OR**

**Select Doctor, Professor, Singer, Actor from ( select  
NameOrder,  
max(case Occupation when 'Doctor' then Name end) as Doctor,  
max(case Occupation when 'Professor' then Name end) as Professor,  
max(case Occupation when 'Singer' then Name end) as Singer,  
max(case Occupation when 'Actor' then Name end) as Actor  
from ( select Occupation, Name, row\_number() over(partition by Occupation order by Name ASC) as NameOrder from Occupations ) as NameLists group by NameOrder  
) as Names**

1. **Binary Tree Nodes**

You are given a table, *BST*, containing two columns: *N*and *P,* where *N* represents the value of a node in *Binary Tree*, and *P* is the parent of *N*.



Write a query to find the node type of *Binary Tree* ordered by the value of the node. Output one of the following for each node:

* *Root*: If node is root node.
* *Leaf*: If node is leaf node.
* *Inner*: If node is neither root nor leaf node.

**Sample Input**



**Sample Output**

1 Leaf

2 Inner

3 Leaf

5 Root

6 Leaf

8 Inner

9 Leaf

**Explanation**

The *Binary Tree* below illustrates the sample:



**Ans:**

**MYSQL**

**SELECT N,**

**CASE**

**WHEN P is NULL THEN 'Root'**

**WHEN N in (SELECT P FROM BST) THEN 'Inner'**

**ELSE 'Leaf'**

**END as node\_type**

**FROM BST**

**ORDER by N;**

1. **New Companies**

Amber's conglomerate corporation just acquired some new companies. Each of the companies follows this hierarchy: 

Given the table schemas below, write a query to print the *company\_code*, *founder* name, total number of *lead* managers, total number of *senior* managers, total number of *managers*, and total number of *employees*. Order your output by ascending *company\_code*.

**Note:**

* The tables may contain duplicate records.
* The *company\_code* is string, so the sorting should not be **numeric**. For example, if the *company\_codes* are *C\_1*, *C\_2*, and *C\_10*, then the ascending *company\_codes* will be *C\_1*, *C\_10*, and *C\_2*.

**Input Format**

The following tables contain company data:

* *Company:* The *company\_code* is the code of the company and *founder* is the founder of the company. 
* *Lead\_Manager:* The *lead\_manager\_code* is the code of the lead manager, and the *company\_code* is the code of the working company. 
* *Senior\_Manager:* The *senior\_manager\_code* is the code of the senior manager, the *lead\_manager\_code* is the code of its lead manager, and the *company\_code* is the code of the working company. 
* *Manager:* The *manager\_code* is the code of the manager, the *senior\_manager\_code* is the code of its senior manager, the *lead\_manager\_code* is the code of its lead manager, and the *company\_code* is the code of the working company. 
* *Employee:* The *employee\_code* is the code of the employee, the *manager\_code* is the code of its manager, the *senior\_manager\_code* is the code of its senior manager, the *lead\_manager\_code* is the code of its lead manager, and the *company\_code* is the code of the working company. 

**Sample Input**

*Company* Table:  *Lead\_Manager* Table:  *Senior\_Manager* Table:  *Manager* Table:  *Employee* Table: 

**Sample Output**

C1 Monika 1 2 1 2

C2 Samantha 1 1 2 2

**Explanation**

In company *C1*, the only lead manager is *LM1*. There are two senior managers, *SM1* and *SM2*, under *LM1*. There is one manager, *M1*, under senior manager *SM1*. There are two employees, *E1* and *E2*, under manager *M1*.

In company *C2*, the only lead manager is *LM2*. There is one senior manager, *SM3*, under *LM2*. There are two managers, *M2* and *M3*, under senior manager *SM3*. There is one employee, *E3*, under manager *M2*, and another employee, *E4*, under manager, *M3*.

**Ans:**

**MYSQL**

**select a.company\_code,a.founder,count(distinct b.lead\_manager\_code), count(distinct b.senior\_manager\_code),count(distinct b.manager\_code),**

**count(distinct b.employee\_code) from**

**Company as a join employee as b**

**on a.company\_code=b.company\_code**

**group by a.company\_code,a.founder**

**order by a.company\_code asc;**

**OR**

**select c.company\_code, c.founder, count(distinct l.lead\_manager\_code),  
count(distinct s.senior\_manager\_code), count(distinct m.manager\_code),  
count(distinct e.employee\_code) from**

**Company as c join Lead\_Manager as l   
on c.company\_code = l.company\_code  
join Senior\_Manager as s  
on l.lead\_manager\_code = s.lead\_manager\_code  
join Manager as m   
on m.senior\_manager\_code = s.senior\_manager\_code  
join Employee as e  
on e.manager\_code = m.manager\_code  
group by c.company\_code, c.founder  
order by c.company\_code;**

* **AGGREGATION**

1. **Revising aggregation – The Count Function**

Query a count of the number of cities in **CITY** having a Population larger than .100000

**Input Format**

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

**Ans:**

**MYSQL**

**SELECT COUNT(NAME) FROM CITY WHERE POPULATION > 100000 ;**

1. **Revising aggregation – The Sum Function**

Query the total population of all cities in **CITY** where District is **California**.

**Input Format**

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

**Ans:**

**select sum(population) from city where district ='California';**

1. **Revising aggregation – Average**

Query the average population of all cities in **CITY** where District is **California**.

**Input Format**

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

**Ans:**

**select avg (population) from city where district = 'California';**

1. **Average Population**

Query the average population for all cities in **CITY**, rounded down to the nearest integer.

**Input Format**

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

**Ans:**

**select floor(avg(population)) from city;**

1. **Japan Population**

Query the sum of the populations for all Japanese cities in **CITY**. The COUNTRYCODE for Japan is **JPN**.

**Input Format**

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

**Ans:**

**select sum(population) from city where countrycode='JPN';**

1. **Population Density Difference**

Query the difference between the maximum and minimum populations in **CITY**.

**Input Format**

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

**Ans:**

**select max(population) - min(population) from city;**

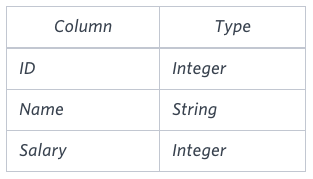
1. **The Blunder**

Samantha was tasked with calculating the average monthly salaries for all employees in the **EMPLOYEES** table, but did not realize her keyboard's  key was broken until after completing the calculation. She wants your help finding the difference between her miscalculation (using salaries with any zeros removed), and the actual average salary.

Write a query calculating the amount of error (i.e.:  average monthly salaries), and round it up to the next integer.

**Input Format**

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

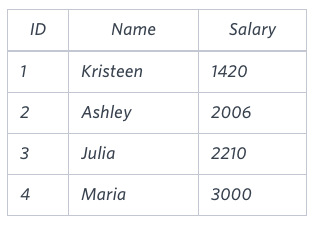


**Note:** Salary is per month.

**Constraints**

.

**Sample Input**

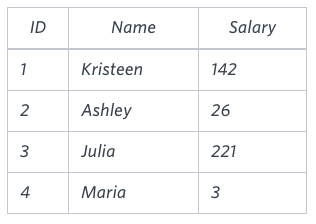


**Sample Output**

2061

**Explanation**

The table below shows the salaries without zeros as they were entered by Samantha:



Samantha computes an average salary of . The actual average salary is .

The resulting error between the two calculations is . Since it is equal to the integer , it does not get rounded up.

**Ans:**

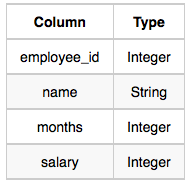
**select max(population) - min(population) from city;**

1. **Top Earners**

We define an employee’s total earnings to be their monthly salary\* months worked, and the maximum total earnings to be the maximum total earnings for any employee in the Employee table. Write a query to find the maximum total earnings for all employees as well as the total number of employees who have maximum total earnings. Then print these values as space-separated integers.

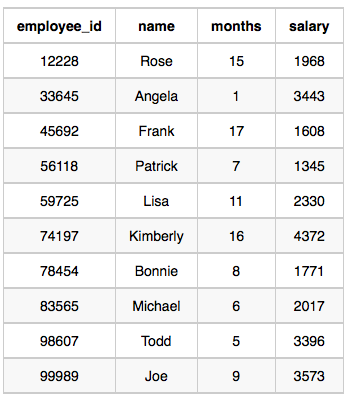
Input Format

The Employee table containing employee data for a company is described as follows:



where employee\_id is an employee’s ID number, name is their name, months is the total number of months they’ve been working for the company, and salary is the their monthly salary.

Sample Input



Sample Output

69952 1

Explanation

The table and earnings data is depicted in the following diagram:



The maximum earnings value is 69952. The only employee with earnings 69952 is Kimberly, so we print the maximum earnings value (69952) and a count of the number of employees who have earned 69952 (which is 1) as two space-separated values.

**Ans:**

**select months\*salary, count(\*) from employee**

**group by months\*salary**

**order by months\*salary desc**

**limit 1;**

1. **Weather Observation Station 2**

Query the following two values from the **STATION** table:

1. The sum of all values in *LAT\_N* rounded to a scale of 2  decimal places.
2. The sum of all values in *LONG\_W* rounded to a scale of  2 decimal places.

**Input Format**

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



where *LAT\_N* is the northern latitude and *LONG\_W* is the western longitude.

**Output Format**

Your results must be in the form:

lat lon

where lat  is the sum of all values in *LAT\_N* and  lon is the sum of all values in *LONG\_W*. Both results must be rounded to a scale of 2  decimal places.

**Ans:**

**select round(sum(LAT\_N),2) AS lat , round(sum(LONG\_W),2) as lon from STATION;**

1. **Weather Observation Station 13**

Query the sum of Northern Latitudes (LAT\_N) from **STATION** having values greater than  38.7880 and less than 137.2345 . Truncate your answer to  4 decimal places.

**Input Format**

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



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Ans:**

**select round(sum(lat\_n),4) from station where lat\_n>38.7880 and lat\_n<137.2345;**

1. **Weather Observation Station 14**

Query the greatest value of the Northern Latitudes (LAT\_N) from **STATION** that is less than 137.345. Truncate your answer to  4 decimal places.

**Input Format**

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



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Ans:**

**SELECT ROUND(MAX(LAT\_N),4) FROM STATION WHERE LAT\_N < 137.345;**

1. **Weather Observation Station 15**

Query the Western Longitude (LONG\_W) for the largest Northern Latitude (LAT\_N) in **STATION** that is less than 137.345. Round your answer to  4 decimal places.

**Input Format**

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



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Ans:**

**SELECT ROUND(LONG\_W,4) FROM STATION WHERE LAT\_N < 137.345**

**ORDER BY LAT\_N DESC LIMIT 1;**

1. **Weather Observation Station 16**

Query the smallest Northern Latitude (LAT\_N) from **STATION** that is greater than . Round your answer to  4 decimal places.

**Input Format**

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



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Ans:**

**SELECT ROUND(MIN(LAT\_N),4) FROM STATION WHERE LAT\_N > 38.7780;**

1. **Weather Observation Station 17**

Query the Western Longitude (LONG\_W)where the smallest Northern Latitude (LAT\_N) in **STATION** is greater than . Round your answer to  decimal places.

**Input Format**

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



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Ans:**

**SELECT ROUND(LONG\_W,4) FROM STATION WHERE LAT\_N>38.7780**

**ORDER BY LAT\_N ASC LIMIT 1;**

1. **Weather Observation Station 18**

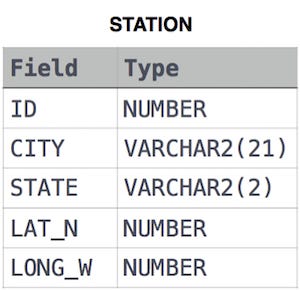
Consider P1(a,b) and P2(c,d) to be two points on a 2D plane.

* a happens to equal the **minimum** value in Northern Latitude (LAT\_N in **STATION**).
* b happens to equal the **minimum** value in Western Longitude (LONG\_W in **STATION**).
* c happens to equal the **maximum** value in Northern Latitude (LAT\_N in **STATION**).
* c happens to equal the **maximum** value in Western Longitude (LONG\_W in **STATION**).

Query the [Manhattan Distance](https://xlinux.nist.gov/dads/HTML/manhattanDistance.html) between points and and round it to a scale of decimal places.

**Input Format**

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



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Ans:**

**/\***

**Manhattan distance**

**Definition: The distance between two points measured along axes at right angles. In a plane with p1 at (x1, y1) and p2 at (x2, y2), it is |x1 - x2| + |y1 - y2|.**

**\*/**

**SELECT ROUND((MAX(LAT\_N) - MIN(LAT\_N))+(MAX(LONG\_W) - MIN(LONG\_W)),4) FROM STATION;**

**OR**

**select ROUND(ABS(MAX(LAT\_N) - MIN(LAT\_N)) + ABS(MAX(LONG\_W) - MIN(LONG\_W)), 4) FROM STATION;**

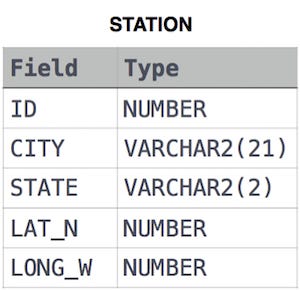
1. **Weather Observation Station 19**

Consider P1(a, c) and P2(b,d) to be two points on a 2D plane where a, c are the respective minimum and maximum values of Northern Latitude (LAT\_N) and b, d are the respective minimum and maximum values of Western Longitude (LONG\_W) in **STATION**.

Query the [Euclidean Distance](https://en.wikipedia.org/wiki/Euclidean_distance) between points and and format your answer to display decimal digits.

**Input Format**

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



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Ans:**

**SELECT ROUND(SQRT(POWER(MAX(LAT\_N)-MIN(LAT\_N),2) + POWER(MAX(LONG\_W)-MIN(LONG\_W),2)),4) FROM STATION;**

1. **Weather Observation Station 20**

A [*median*](https://en.wikipedia.org/wiki/Median) is defined as a number separating the higher half of a data set from the lower half. Query the median of the Northern Latitudes (LAT\_N) from **STATION** and round your answer to  decimal places.

**Input Format**

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



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Ans:**

**SET @r = 0;**

**SELECT ROUND(AVG(Lat\_N), 4)**

**FROM (SELECT (@r := @r + 1) AS r, Lat\_N FROM Station ORDER BY Lat\_N) Temp**

**WHERE**

**r = (SELECT CEIL(COUNT(\*) / 2) FROM Station) OR**

**r = (SELECT FLOOR((COUNT(\*) / 2) + 1) FROM Station);**

**OR**

**SELECT ROUND(t.lat\_n,4) FROM (SELECT lat\_n, NTILE(2) OVER (ORDER BY lat\_n) as median FROM station) t WHERE median = 1 ORDER BY lat\_n DESC LIMIT 1;**

* **BASIC JOIN**

1. **POPULATION SENSUS**

Given the **CITY** and **COUNTRY** tables, query the sum of the populations of all cities where the CONTINENT is 'Asia'.

**Note:** CITY.CountryCode and COUNTRY.Code are matching key columns.

**Input Format**

The **CITY** and **COUNTRY** tables are described as follows: 



**Ans:**

**SELECT SUM(A.POPULATION) FROM**

**CITY AS A INNER JOIN COUNTRY AS B**

**ON A.COUNTRYCODE=B.CODE**

**WHERE B.CONTINENT='ASIA';**

1. **AFRICAN CITIES**

Given the **CITY** and **COUNTRY** tables, query the names of all cities where the CONTINENT is 'Africa'.

**Note:** CITY.CountryCode and COUNTRY.Code are matching key columns.

**Input Format**

The **CITY** and **COUNTRY** tables are described as follows: 



**Ans:**

**SELECT A.NAME FROM**

**CITY AS A INNER JOIN COUNTRY AS B**

**ON A.COUNTRYCODE=B.CODE**

**WHERE CONTINENT='AFRICA';**

1. **AVERAGE POPULATION OF EACH CONTINENT**

Given the **CITY** and **COUNTRY** tables, query the names of all the continents (COUNTRY.Continent) and their respective average city populations (CITY.Population) rounded down to the nearest integer.

**Note:** CITY.CountryCode and COUNTRY.Code are matching key columns.

**Input Format**

The **CITY** and **COUNTRY** tables are described as follows: 



**Ans:**

**SELECT B.CONTINENT,FLOOR(AVG(A.POPULATION)) FROM**

**CITY AS A INNER JOIN COUNTRY AS B**

**ON A.COUNTRYCODE=B.CODE**

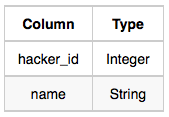
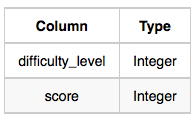
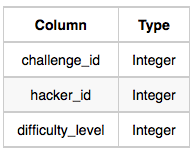
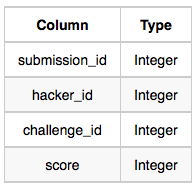
**GROUP BY CONTINENT;**

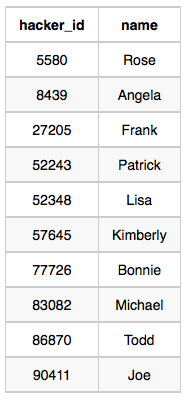
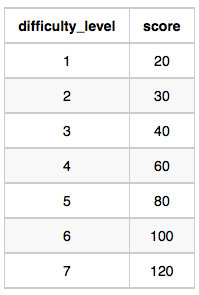
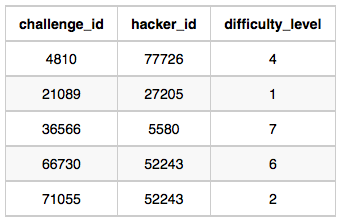
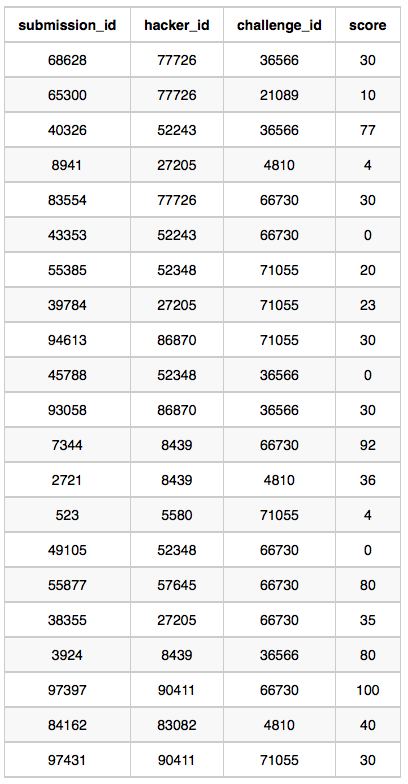
1. **TOP COMPETITORS**

Julia just finished conducting a coding contest, and she needs your help assembling the leaderboard! Write a query to print the respective *hacker\_id* and *name* of hackers who achieved full scores for *more than one* challenge. Order your output in descending order by the total number of challenges in which the hacker earned a full score. If more than one hacker received full scores in same number of challenges, then sort them by ascending *hacker\_id*.

**Input Format**

The following tables contain contest data:

* *Hackers:* The *hacker\_id* is the id of the hacker, and *name* is the name of the hacker. 
* *Difficulty:* The *difficult\_level* is the level of difficulty of the challenge, and *score* is the score of the challenge for the difficulty level. 
* *Challenges:* The *challenge\_id* is the id of the challenge, the *hacker\_id* is the id of the hacker who created the challenge, and *difficulty\_level* is the level of difficulty of the challenge. 
* *Submissions:* The *submission\_id* is the id of the submission, *hacker\_id* is the id of the hacker who made the submission, *challenge\_id* is the id of the challenge that the submission belongs to, and *score* is the score of the submission. 

**Sample Input***Hackers* Table:  *Difficulty* Table:  *Challenges* Table:  *Submissions* Table: 

**Sample Output**

90411 Joe

**Explanation**

Hacker *86870* got a score of *30* for challenge *71055* with a difficulty level of *2*, so *86870* earned a full score for this challenge.

Hacker *90411* got a score of *30* for challenge *71055* with a difficulty level of *2*, so *90411* earned a full score for this challenge.

Hacker *90411* got a score of *100* for challenge *66730* with a difficulty level of *6*, so *90411* earned a full score for this challenge.

Only hacker *90411* managed to earn a full score for more than one challenge, so we print the their *hacker\_id* and *name* as  2 space-separated values.

**Ans:**

**SELECT S.hacker\_id, name**

**FROM SUBMISSIONS AS S**

**JOIN HACKERS AS H ON S.hacker\_id = H.hacker\_id**

**JOIN Challenges AS C ON S.challenge\_id = C.challenge\_id**

**JOIN Difficulty AS D ON C.difficulty\_level = D.difficulty\_level**

**WHERE S.score = D.score**

**GROUP BY name, S.hacker\_id**

**HAVING count(S.challenge\_id) > 1**

**ORDER BY count(S.challenge\_id) DESC, S.hacker\_id ASC ;**

1. **OLLIVANDER’S INVENTORY**

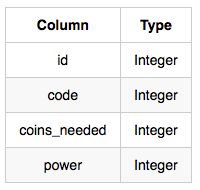
Harry Potter and his friends are at Ollivander’s with Ron, finally replacing Charlie’s old broken wand.

Hermione decides the best way to choose is by determining the minimum number of gold galleons needed to buy each non-evil wand of high power and age. Write a query to print the id, age, coins\_needed, and power of the wands that Ron’s interested in, sorted in order of descending power. If more than one wand has same power, sort the result in order of descending age.

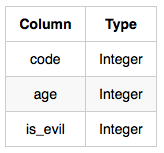
Input Format

The following tables contain data on the wands in Ollivander’s inventory:

* Wands: The id is the id of the wand, code is the code of the wand, coins\_needed is the total number of gold galleons needed to buy the wand, and power denotes the quality of the wand (the higher the power, the better the wand is).

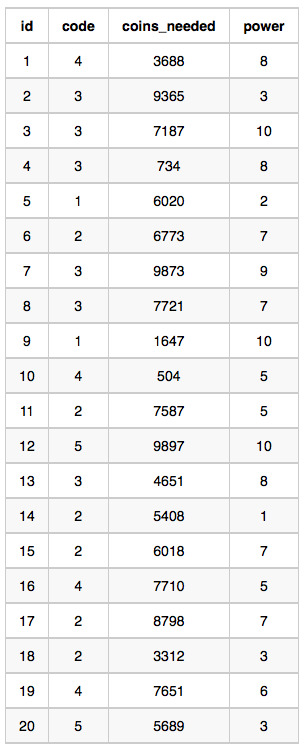


* Wands\_Property: The code is the code of the wand, age is the age of the wand, and is\_evil denotes whether the wand is good for the dark arts. If the value of is\_evil is 0, it means that the wand is not evil. The mapping between code and age is one-one, meaning that if there are two pairs, and , then and .

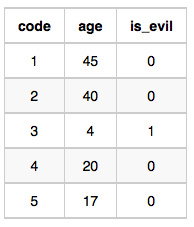


Sample Input

Wands Table:



Wands\_Property Table:



Sample Output

9 45 1647 10  
12 17 9897 10  
1 20 3688 8  
15 40 6018 7  
19 20 7651 6  
11 40 7587 5  
10 20 504 5  
18 40 3312 3  
20 17 5689 3  
5 45 6020 2  
14 40 5408 1

**Ans:**

**select w.id, p.age, w.coins\_needed, w.power from Wands as w**

**join Wands\_Property as p**

**on w.code = p.code**

**where w.coins\_needed = (select min(coins\_needed)**

**from Wands w2 inner join Wands\_Property p2**

**on w2.code = p2.code**

**where p2.is\_evil = 0 and p.age = p2.age and w.power = w2.power)**

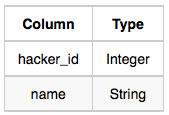
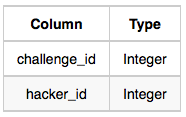
**order by w.power desc, p.age desc;**

1. **CHALLENGES**

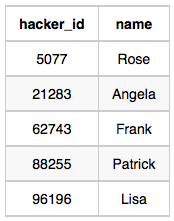
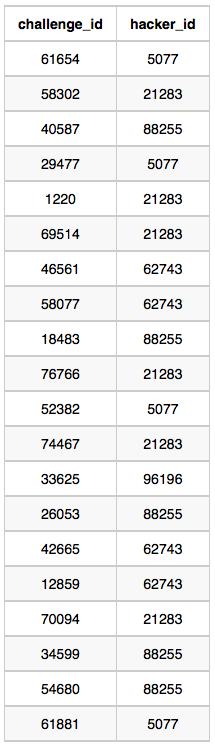
Julia asked her students to create some coding challenges. Write a query to print the *hacker\_id*, *name*, and the total number of challenges created by each student. Sort your results by the total number of challenges in descending order. If more than one student created the same number of challenges, then sort the result by *hacker\_id*. If more than one student created the same number of challenges and the count is less than the maximum number of challenges created, then exclude those students from the result.

**Input Format**

The following tables contain challenge data:

* *Hackers:* The *hacker\_id* is the id of the hacker, and *name* is the name of the hacker. 
* *Challenges:* The *challenge\_id* is the id of the challenge, and *hacker\_id* is the id of the student who created the challenge. 

**Sample Input 0**

*Hackers* Table:  *Challenges* Table: 

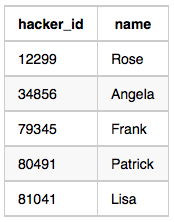
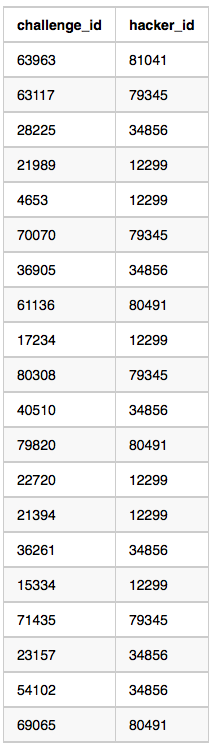
**Sample Output 0**

21283 Angela 6

88255 Patrick 5

96196 Lisa 1

**Sample Input 1**

*Hackers* Table:  *Challenges* Table: 

**Sample Output 1**

12299 Rose 6

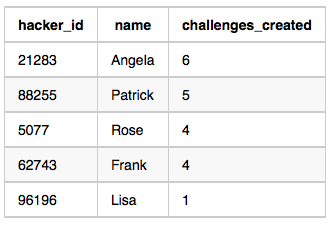
34856 Angela 6

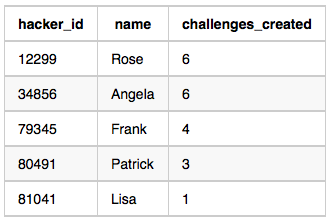
79345 Frank 4

80491 Patrick 3

81041 Lisa 1

**Explanation**

For *Sample Case 0*, we can get the following details:  
  
Students  and  both created  challenges, but the maximum number of challenges created is  so these students are excluded from the result.

For *Sample Case 1*, we can get the following details:  
  
Students  and  both created  challenges. Because  is the maximum number of challenges created, these students are included in the result.

**Ans:**

**SELECT H.hacker\_id, H.name,COUNT(C.challenge\_id) AS CNT FROM**

**HACKERS AS H JOIN CHALLENGES AS C**

**ON H.HACKER\_ID=C.HACKER\_ID**

**GROUP BY H.hacker\_id, H.name**

**HAVING CNT=(SELECT COUNT(C1.challenge\_id) FROM CHALLENGES AS C1 GROUP BY c1.hacker\_id ORDER BY count(\*) desc limit 1)**

**OR**

**CNT NOT IN (SELECT COUNT(C2.challenge\_id) FROM CHALLENGES AS C2 GROUP BY c2.hacker\_id HAVING c2.hacker\_id <> H.hacker\_id)**

**ORDER BY COUNT(C.challenge\_id) DESC,H.HACKER\_ID ASC;**

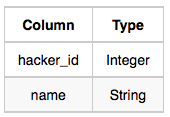
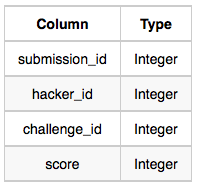
1. **CONTEST LEADERBOARD**

You did such a great job helping Julia with her last coding contest challenge that she wants you to work on this one, too!

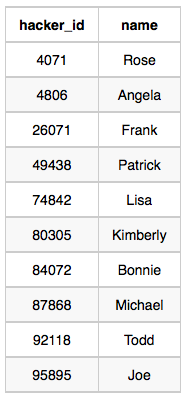
The total score of a hacker is the sum of their maximum scores for all of the challenges. Write a query to print the hacker\_id, name, and total score of the hackers ordered by the descending score. If more than one hacker achieved the same total score, then sort the result by ascending hacker\_id. Exclude all hackers with a total score of  0 from your result.

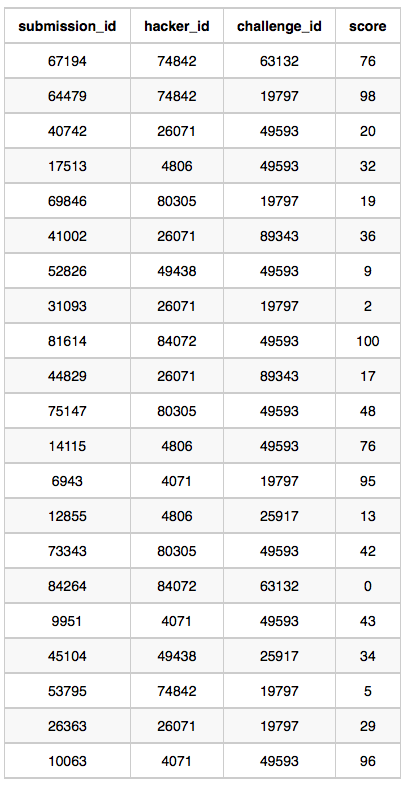
**Input Format**

The following tables contain contest data:

* Hackers: The hacker\_id is the id of the hacker, and name is the name of the hacker. 
* Submissions: The submission\_id is the id of the submission, hacker\_id is the id of the hacker who made the submission, challenge\_id is the id of the challenge for which the submission belongs to, and score is the score of the submission. 

**Sample Input**

Hackers Table: 

Submissions Table: 

**Sample Output**

4071 Rose 191

74842 Lisa 174

84072 Bonnie 100

4806 Angela 89

26071 Frank 85

80305 Kimberly 67

49438 Patrick 43

**Explanation**

Hacker 4071 submitted solutions for challenges 19797 and 49593, so the total score .

Hacker 74842 submitted solutions for challenges 19797 and 63132, so the total score

Hacker 84072 submitted solutions for challenges 49593 and 63132, so the total score .

The total scores for hackers 4806, 26071, 80305, and 49438 can be similarly calculated.

**Ans:**

**select m.hacker\_id, h.name, sum(score) as total\_score from**

**(select hacker\_id, challenge\_id, max(score) as score**

**from Submissions group by hacker\_id, challenge\_id) as m**

**join Hackers as h**

**on m.hacker\_id = h.hacker\_id**

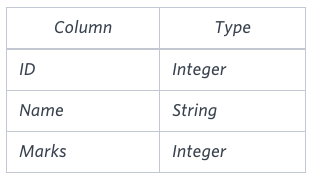
**group by m.hacker\_id, h.name**

**having total\_score > 0**

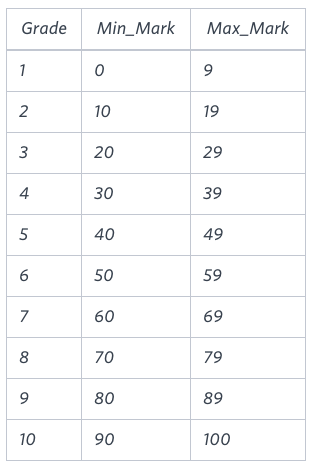
**order by total\_score desc, m.hacker\_id;**

1. **The Report**

You are given two tables: *Students* and *Grades*. *Students* contains three columns *ID*, *Name* and *Marks*.



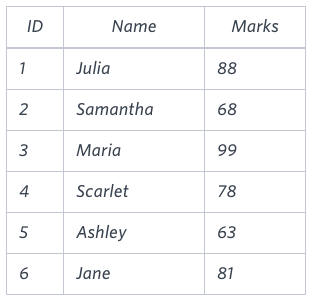
*Grades* contains the following data:



*Ketty* gives *Eve* a task to generate a report containing three columns: *Name*, *Grade* and *Mark*. *Ketty* doesn't want the NAMES of those students who received a grade lower than *8*. The report must be in descending order by grade -- i.e. higher grades are entered first. If there is more than one student with the same grade (8-10) assigned to them, order those particular students by their name alphabetically. Finally, if the grade is lower than 8, use "NULL" as their name and list them by their grades in descending order. If there is more than one student with the same grade (1-7) assigned to them, order those particular students by their marks in ascending order.

Write a query to help Eve.

**Sample Input**



**Sample Output**

Maria 10 99

Jane 9 81

Julia 9 88

Scarlet 8 78

NULL 7 63

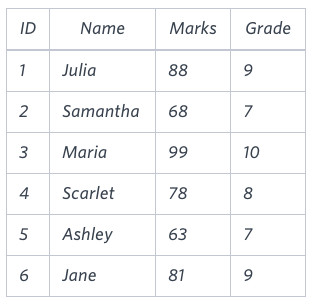
NULL 7 68

**Note**

Print "NULL"  as the name if the grade is less than 8.

**Explanation**

Consider the following table with the grades assigned to the students:



So, the following students got *8*, *9* or *10* grades:

* *Maria (grade 10)*
* *Jane (grade 9)*
* *Julia (grade 9)*
* *Scarlet (grade 8)*

**Ans:**

**SELECT**

**CASE**

**WHEN GRADES.GRADE >= 8 THEN STUDENTS.NAME**

**WHEN GRADES.GRADE < 8 THEN NULL**

**END AS NAME,**

**GRADES.GRADE, STUDENTS.MARKS**

**FROM STUDENTS**

**LEFT JOIN GRADES**

**ON STUDENTS.MARKS >= MIN\_MARK AND STUDENTS.MARKS <= MAX\_MARK**

**ORDER BY**

**GRADES.GRADE DESC, STUDENTS.NAME ASC, STUDENTS.MARKS ASC;**

**OR**

**SELECT (CASE WHEN GRADE <8 THEN 'NULL' ELSE NAME END) AS NAME, GRADE, MARKS**

**FROM (**

**SELECT ID, NAME, MARKS,(CASE**

**WHEN MARKS >= 0 AND MARKS <= 9 THEN 1**

**WHEN MARKS >= 10 AND MARKS <= 19 THEN 2**

**WHEN MARKS >= 20 AND MARKS <= 29 THEN 3**

**WHEN MARKS >= 30 AND MARKS <= 39 THEN 4**

**WHEN MARKS >= 40 AND MARKS <= 49 THEN 5**

**WHEN MARKS >= 50 AND MARKS <= 59 THEN 6**

**WHEN MARKS >= 60 AND MARKS <= 69 THEN 7**

**WHEN MARKS >= 70 AND MARKS <= 79 THEN 8**

**WHEN MARKS >= 80 AND MARKS <= 89 THEN 9**

**ELSE 10 END) AS GRADE**

**FROM STUDENTS)**

**ORDER BY GRADE DESC,**

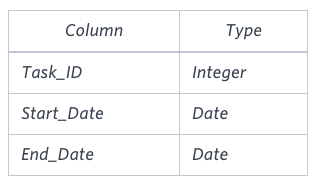
**CASE WHEN GRADE >= 8 AND GRADE <= 10 THEN GRADE END DESC,**

**CASE WHEN GRADE >= 8 AND GRADE <= 10 THEN NAME END ASC,**

**CASE WHEN GRADE >= 1 AND GRADE <= 7 THEN MARKS END ASC;**

1. **SQL Project Planning**

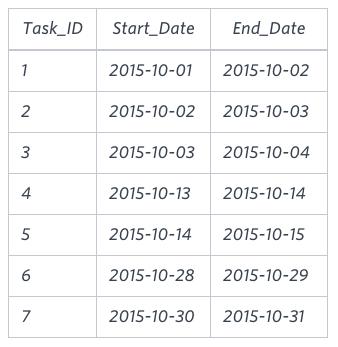
You are given a table, *Projects*, containing three columns: *Task\_ID*, *Start\_Date* and *End\_Date*. It is guaranteed that the difference between the *End\_Date* and the *Start\_Date* is equal to *1* day for each row in the table.



If the *End\_Date* of the tasks are consecutive, then they are part of the same project. Samantha is interested in finding the total number of different projects completed.

Write a query to output the start and end dates of projects listed by the number of days it took to complete the project in ascending order. If there is more than one project that have the same number of completion days, then order by the start date of the project.

**Sample Input**



**Sample Output**

2015-10-28 2015-10-29

2015-10-30 2015-10-31

2015-10-13 2015-10-15

2015-10-01 2015-10-04

**Explanation**

The example describes following *four* projects:

* *Project 1*: Tasks *1*, *2* and *3* are completed on consecutive days, so these are part of the project. Thus start date of project is *2015-10-01* and end date is *2015-10-04*, so it took *3 days* to complete the project.
* *Project 2*: Tasks *4* and *5* are completed on consecutive days, so these are part of the project. Thus, the start date of project is *2015-10-13* and end date is *2015-10-15*, so it took *2 days* to complete the project.
* *Project 3*: Only task *6* is part of the project. Thus, the start date of project is *2015-10-28* and end date is *2015-10-29*, so it took *1 day* to complete the project.
* *Project 4*: Only task *7* is part of the project. Thus, the start date of project is *2015-10-30* and end date is *2015-10-31*, so it took *1 day* to complete the project.

**Ans:**

**SELECT START\_DATE, MIN(END\_DATE)**

**FROM**

**(SELECT START\_DATE**

**FROM PROJECTS**

**WHERE START\_DATE NOT IN**

**(SELECT END\_DATE**

**FROM PROJECTS)) A,**

**(SELECT END\_DATE**

**FROM PROJECTS**

**WHERE END\_DATE NOT IN**

**(SELECT START\_DATE**

**FROM PROJECTS)) B**

**WHERE START\_DATE < END\_DATE**

**GROUP BY START\_DATE**

**ORDER BY (MIN(END\_DATE) - START\_DATE), START\_DATE;**

1. **SQL Project Planning**
2. **SQL Project Planning**
3. **SQL Project Planning**

* **ALTERNATIVE QUERIES**

1. **Print Prime Numbers**

Write a query to print all prime numbers less than or equal to 1000. Print your result on a single line, and use the ampersand (&) character as your separator (instead of a space).

For example, the output for all prime numbers  <= 10 would be:

2&3&5&7

**Ans:**

**SELECT GROUP\_CONCAT(NUMB SEPARATOR '&')**

**FROM (**

**SELECT @num:=@num+1 as NUMB FROM**

**information\_schema.tables t1,**

**information\_schema.tables t2,**

**(SELECT @num:=1) tmp**

**) tempNum**

**WHERE NUMB<=1000 AND NOT EXISTS(**

**SELECT \* FROM (**

**SELECT @nu:=@nu+1 as NUMA FROM**

**information\_schema.tables t1,**

**information\_schema.tables t2,**

**(SELECT @nu:=1) tmp1**

**LIMIT 1000**

**) tatata**

**WHERE FLOOR(NUMB/NUMA)=(NUMB/NUMA) AND NUMA<NUMB AND NUMA>1**

**) ;**

1. **Draw the Triangle 1**

P(R) represents a pattern drawn by Julia in R rows. The following pattern represents P(5):

\* \* \* \* \*

\* \* \* \*

\* \* \*

\* \*

\*

Write a query to print the pattern P(20).

**Ans:**

**set @number = 20+1;**

**select repeat('\* ', @number := @number - 1)**

**from information\_schema.tables ;**

1. **Draw the Triangle 2**

P(R) represents a pattern drawn by Julia in R rows. The following pattern represents P(5):

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

Write a query to print the pattern P(20).

**Ans:**

**set @number = 0;**

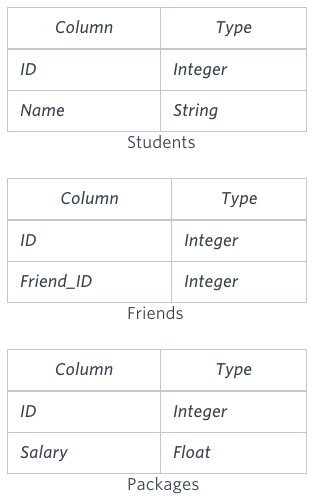
**select repeat('\* ', @number := @number + 1)**

**from information\_schema.tables**

**where @number < 20 ;**

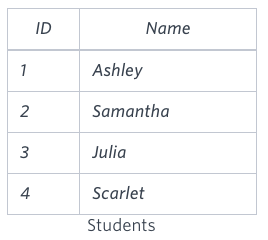
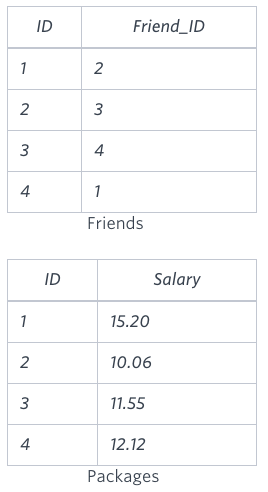
1. **Placement**

You are given three tables: *Students*,*Friends*and*Packages.* *Students* contains two columns: *ID* and *Name*. *Friends* contains two columns: *ID* and *Friend\_ID* (*ID* of the ONLY best friend). *Packages* contains two columns: *ID* and *Salary* (offered salary in $ thousands per month).



Write a query to output the names of those students whose best friends got offered a higher salary than them. Names must be ordered by the salary amount offered to the best friends. It is guaranteed that no two students got same salary offer.

**Sample Input**

**Sample Output**

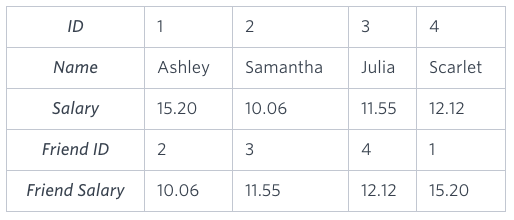
Samantha

Julia

Scarlet

**Explanation**

See the following table:



Now,

* *Samantha's* best friend got offered a higher salary than her at 11.55
* *Julia's* best friend got offered a higher salary than her at 12.12
* *Scarlet's* best friend got offered a higher salary than her at 15.2
* *Ashley's* best friend did NOT get offered a higher salary than her

The name output, when ordered by the salary offered to their friends, will be:

* *Samantha*
* *Julia*
* *Scarlet*

**Ans:**

**SELECT t.Name**

**FROM (**

**SELECT s1.ID, s1.Name, p1.Salary, f.Friend\_ID, s2.name as friend\_name, p2.Salary as friend\_salary**

**FROM Students s1**

**JOIN Packages p1 ON s1.ID = p1.ID**

**JOIN Friends f ON s1.ID = f.ID**

**JOIN Students s2 ON f.Friend\_ID = s2.ID**

**JOIN Packages p2 ON f.Friend\_ID = p2.ID**

**) t**

**WHERE t.friend\_salary > t.Salary**

**ORDER BY friend\_salary;**

1. **Placement**