The 2016 ACM ICPC Asia Regional Contest Dhaka Site

Sponsored by IBM
Hosted by UAP
Dhaka, Bangladesh

19th November 2016
You get 19 Pages
11 Problems &
300 Minutes

acm International Collegiate Programming Contest

IBM event sponsor

Judges & Problemsetters

Name	Affiliation
Shahriar Manzoor	Southeast University, Bangladesh & CodeMarshal
Mohammad Mahmudur Rahman	CodeMarshal & Mukto Software Ltd
Md. Shiplu Hawlader	University of Dhaka
Md Mahbubul Hasan	Google Inc
Derek Kisman	Translattice, USA
Rujia Liu	Eryiju, China
Monirul Hasan	Southeast University, Bangladesh
Kazi Rakibul Hossain	Dynamic Solvers Bangladesh
Anindya Das	Iowa State University, USA
Mahafuzur Rahman	CodeMarshal & Mukto Software Ltd
Hasnain Heickal	University of Dhaka
Muhammed Hedayetul Islam	Google Inc
Shafaet Ashraf	HackerRank
Kaysar Abdullah	Bangladesh University of Engineering & Technology
Syed Shahriar Manjur	Google Inc

Rules for ACM-ICPC 2016 Asia Regional Dhaka Site Onsite Contest:

a) Solutions to problems submitted for judging are called runs. Each run is judged as accepted or rejected by the judge, and the team is notified of the results.

b) Notification of accepted and rejected runs will continue until the end of the contest.

c) A contestant may submit a clarification request to judges only through the CodeMarshal clarification system. If the judges agree that an ambiguity or error exists, a clarification will be issued to all contestants. Judges may prefer not to answer a clarification at all, in which case that particular clarification request will be marked as IGNORED in the CodeMarshal clarification page.

d) Contestants are not to converse with anyone except members of their team and personnel designated by the organizing committee while seated at the team desk. **They cannot even talk with their team members when they are walking around the contest floor to have food or any other purpose.** System support staffs or judges may advise contestants on system-related problems such as explaining system error messages.

e) While the contest is scheduled for a particular time length (five hours), the contest director has the authority to alter the length of the contest in the event of unforeseen difficulties. Should the contest duration be altered, every attempt will be made to notify contestants in a timely and uniform manner.

f) **A team may be disqualified by the Contest Director** for any activity that jeopardizes the contest such as dislodging extension cords, unauthorized modification of contest materials, distracting behavior or communicating with other teams. The **judges on the contest floor** will report to the **Judging Director** about distracting behavior of any team. **The judges can also recommend penalizing a team with additional penalty minutes for their distracting behavior.**

g) Nine, ten, eleven or twelve problems will be posed. So far as possible, problems will avoid dependence on the detailed knowledge of a particular application area or particular contest language. Of these problems at least one will be solvable by a first year computer science student, another one will be solvable by a second year computer science student and rest will determine the winner.

h) Contestants will have foods available in their contest room during the contest. So they cannot leave the contest room during the contest without explicit permission from the invigilators. **The contestants are not allowed to communicate with any contestant (even contestants of his own team) or coaches when they are outside the contest arena.**

i) Teams can bring up to **200 pages of printed materials** with them and they can also bring five additional books. But they are not allowed to bring calculators or any machine-readable devices like CD, DVD, Pen-drive, IPOD, MP3/MP4 players, floppy disks etc. **Mobile phone MUST be switched off at all times and stored inside a bag or any other place that is publicly non visible during the entire contest time. Failure to adherence to this clause under any condition will very likely lead to strict disciplinary retaliation and possible disqualification.**

j) With the help of the volunteers, the contestants can have printouts of their codes for debugging purposes. **Passing of printed codes to other teams is strictly prohibited.**

k) The decision of the judges is final.

l) Teams should inform the volunteers/judges if they don't get verdict from the codemarshal within 5 minutes of submission. Teams should also notify the volunteers if they cannot login into the CodeMarshal system. These sort of complains will not be entertained after the contest.

A

A Giveaway

Input: Standard Input
Output: Standard Output

A positive integer number is "Special" if it is both a square (eg. **1, 4, 9, 16, 64 ...**) and a cube (eg. **1, 8, 27, 64 ...**). The smallest special number is **1**. Now your job is to write a program that finds whether a number less than **100000000** is special or not. It may be noted that there are only **21** such numbers within this range and these are **1, 64, 729, 4096, 15625, 46656, 117649, 262144, 531441, 1000000, 1771561, 2985984, 4826809, 7529536, 11390625, 16777216, 24137569, 34012224, 47045881, 64000000** and **85766121**. A very childish but legitimate C/C++ solution, which would work for positive numbers not exceeding **15624**, is shown below.

```
#include<stdio.h>
int main(void)
{
      int num;
       while(scanf("%d",&num)  && num)
      {
            if(num==1 || num==64 || num==729 || num==4096)
                printf("Special\n");
            else
                printf("Ordinary\n");
      }
       return 0;
}
```

A C/C++ code that will work for positive numbers not exceeding 15624

Input

The input file contains at most **1001** lines of input. Each line contains a positive integer less than **100000000**. Input is terminated by a line containing a zero.

Output

For each line of input except the last one produce one line of output. This line contains a string (without the quotes) **"Special"** if the number is special and **"Ordinary"** if the number is not special. Look at the output for the sample input for details.

Sample Input

```
1
2
64
100
15625
0
```

Output for Sample Input

```
Special
Ordinary
Special
Ordinary
Special
```

B Game of XOR

Input: Standard Input
Output: Standard Output

I assume you know about the XOR operation. In case you don't know, XOR operation of two binary bits is **0** if they are equal, otherwise **1**. That means,

0 XOR 0	0
0 XOR 1	1
1 XOR 0	1
1 XOR 1	0

Initially you will be given a binary string (a string consisting of only **0** or **1**). This will be the first generation. You can generate the **(i+1)**'th generation from the i'th generation by inserting XOR of adjacent bits. So if the i'th generation is **01001** then, the **(i+1)**'th generation would be **0(1)1(1)0(0)0(1)1** or **011100011** (the bits in the parenthesis are **XOR** of their adjacent bits).

One more example for more than one generations:

First generation	1 0 0 1 0
Second generation	1 1 0 0 0 1 1 1 0
Third generation	1 0 1 1 0 0 0 0 0 1 1 0 1 0 1 1 0
Fourth generation	1101101100000000011011011101101 10

The spaces are put in the above table for the sake of clarity and to show you how the input is formatted.

You will be given a binary string **S**, a positive integer **G** and two positions at the **G**'th generation of **S** (original binary string **S** is the first generation), you need to find the number of **0**s and **1**s between the given two positions (inclusively).

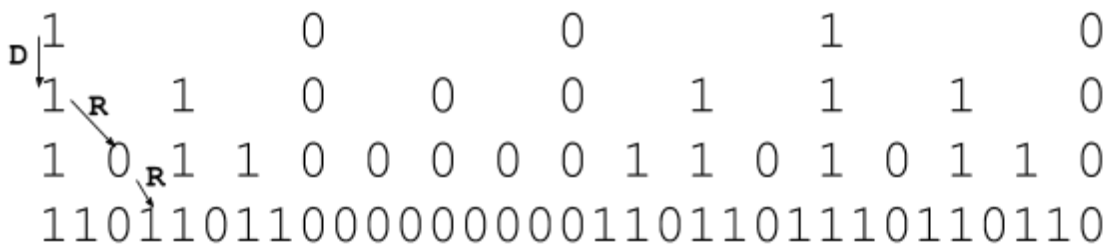
Since the **G**'th generation would be very long, the position will be defined by: **p T**, where **p** is an integer (**0 ≤ p <** the length of **S**) and **T** is a string of length **(G - 1)** consisting of only **D** (down), or/and **R** (right). To get the position at the **G**'th generation, you will start at the **p**'th position of the first generation and move according to the command in **T**.

Suppose you are at the **i**'th generation and the **j**'th character (**0**-indexed) in that generation (so initially, **i = 1, j = p**).
If the next command is **D**, new **i = i + 1** (move to the next generation), new **j = 2 * j** (move exactly down according to the above table).
If the next command is **R**, new **i = i + 1** (move to the next generation), new **j = 2 * j + 1** (move right in the next generation, according to the above table).

For details please see the table below:

p T	The position (bold) at the **G**'th generation denoted by "**p T**" of 10010
0 DRR	110**1**10110000000001101101110110110
2 RRR	11011011000000000110110**1**110110110
1 RDD	11011011000**0**0000110110111011011O

Below the "0 DRR" case is shown.



Input

The first line of the input contains an integer **T (T ≤ 50)** which denotes the number of test cases. **T** test cases follow.

There are three lines in a case. First line contains **S** (**1** ≤ the length of **S** ≤ **100000**) and **G (1** ≤ **G** ≤ **10000)**. Next two lines contain two positions in the **G**'th generation denoted by "p T" (**0** ≤ **p <** the length of **S** and The length of T = **G - 1**) as above. You may assume that **"p T"** is a valid input (i.e. it won't be like: **p = (|S| - 1)** and **T = R...** since there is nothing to the right of the last character). You may also assume that the position denoted in the second line would not be beyond the position denoted in the third line.

Output

For each test case, output case number followed by number of **0**s and **1**s. Since number of **0**s and **1**s would be large, you need to output them by modulo **342307123**. For details of the input/output format please look at the sample input/output.

Sample Input

```
2
10010 4
0 DRR
1 RDD
01001 2
0 D
4 D
```

Output for Sample Input

```
Case 1: 6 4
Case 2: 4 5
```

C National Bomb Defusing Squad

Input: Standard Input
Output: Standard Output

You are an IT and Statistics specialist in the National Bomb Defusing Squad. You have to use your programming and statistical skills to find out locations of probable suicide bomb attacks. Sometimes you also have to find the blast radius of bombs from the death toll. In this specific problem, you will have to find the expected number of deaths due to a suicide bomb attack in a crowded place. For simplicity, in this problem you can assume:

(1) Each person can be considered as a point in a **2-D** Cartesian plane. So an **(x, y)** coordinate can be used to denote the location of a person. Here **x** and **y** are always non-negative integers.
(2) All the persons in a scenario are equally likely to carry a bomb. But exactly one person will carry the bomb.
(3) More than one person can be at the same coordinate.
(4) A suicide bomb has a blasting radius **R**. Everyone within the blasting radius dies. For example a person at location **(p, q)** is carrying a bomb and there is another person at **(m, n)**. When the bomb explodes at **(p, q)** the person at **(m, n)** will die if his distance from (p, q) is not more than **R**. The problem for you to solve is - for a given scenario you will have to calculate the death toll for up to thousand values of R.

Input

The input file contains maximum **7** test cases.

First line of each test case contains two integers **N (0 < N ≤ 3000)** and **Q (0 < Q ≤ 1000)**. Here **N** denotes the number of people in the scenario and **Q** denotes the number of queries to follow. Each of the next **N** lines contains two integers **(x_i, y_i)** that denotes the Cartesian coordinate of one person in the scenario. These integers are non negative and do not exceed **25000**. Each of the next **Q** lines contains a single integer **R_j ($0<R_j<40001$).** Input is terminated by a line containing two zeroes.

Output

For each set of input produce **Q** lines of outputs. Each of this **Q** lines contains output for one query - The expected number of deaths if the blast radius of the bomb is **R_j**. This value should be rounded to two digits after the decimal point. **Print a blank line after the output for each test case.**

Sample Input	Output for Sample Input
4 3 1 1 1 2 12 3 40 40 1 10 100 0 0	1.50 1.50 4.00

D

Rational Grading

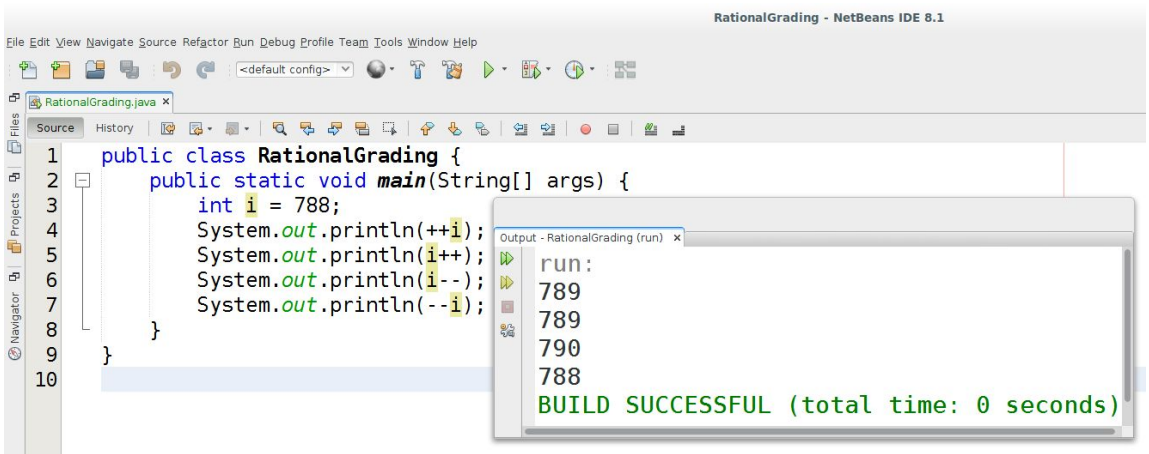
Input: Standard Input
Output: Standard Output

Grading exam scripts of a programming course is often a pain and to reduce the pain teachers often give full marks only when the examinee's output matches exactly with the correct one. Then the mark can be easily calculated as a percentage of matching. But in some scenario, this approach may not be right - especially in those cases where the output of each step depends on that of the previous one. For example in the figure below you can see a program and its correct output. The 2nd, 3rd and 4th output depends on the output of the previous lines. So if the output of the first line is wrong and the next three lines are correct the examinee should not get **3** marks because he has made mistakes in all four cases (Or made a mistake while copying from his friend).

```cpp
#include<iostream>
using namespace std;
int main(void)
{
    int i=788;
    cout<<++i<<endl;
    cout<<i++<<endl;
    cout<<i--<<endl;
    cout<<--i<<endl;
    return 0;
}
```

```
C:\TurboC++\Disk\TurboC3\BIN\arch2016\grading.exe
789
789
790
788

Process returned 0 (0x0)   execution time :
Press any key to continue.
```



A simple C++ and Java Program on the left and corresponding correct output on the right

So to grade such a program we adopt the following new policy - we should judge the correctness of each line based on the output of the **immediate previous** line. For example, if for the program above if someone's output is:

1000
1000
1001
999

He/she should get **3** marks. Because, the output of the first line is incorrect but based on that line's output, the output of next three lines are correct.

Given a program's description and its output, you job is to calculate the mark that the examinee will get.

Input

There are at most 1001 test cases. The description of each test case is given below:

Each test case starts with two integers **ivalue (0 < ivalue$_{10}$ * 1000000$_{10}$)** and **t (0 < t * 30)**. Here **ivalue** denotes the initial value of the variable **i**. This value can be in decimal (leftmost digit is not zero), octal (leftmost digit is zero) or hexadecimal (starts with 0x and non-numeric digits will be in uppercase). **t** denotes the total number of printing statements in the program. Each of the next **t** lines contains an expression that the printing function will print followed by the output for the expression written by the examinee. The expressions can be anyone from the following instruction set, **s = {i++, ++i, i--, --i, i},** the output value will be between 0 and 1000000 (inclusive) and of course in decimal.

Input is terminated by a line containing two zeroes.

Output

For each test case produce one line of output. This line contains the mark that the student will get according to the new policy.

Sample Input

```
766 4
++i 767
i++ 767
i-- 768
--i 766
0766 4
++i 789
i++ 789
i-- 790
--i 788
0x766 4
++i 1895
i++ 1895
i-- 1896
--i 1894
0 0
```

Output for Sample Input

```
4
3
4
```

Illustration of 2nd Sample Input

Expression	Current Value of i before printing based on ivalue$_{10}$ or the value in the previous line	Printed Value (By student)	Correct Expected Value	Marks obtained	Value of i after being printed
++i	0766 = 502	ĄąĆ	ĂýĀ	ý	789
i++	789	789	789	1	790
i--	790	790	790	1	789
--i	789	788	788	1	788
Total Marks obtained				3	

E Balanced String

Input: Standard Input
Output: Standard Output

A string consisting of parenthesis **"("** and **")"** is called **balanced string** if any of the following is true.

a. If the string is empty.
b. If **A** and **B** are balanced string, **AB** is also balanced string.
c. If **A** is a balanced string, **(A)** is also a balanced string

Now from a balanced string we can generate a sequence of numbers. The sequence is generated as follows:

- Start scanning the string from left to right and after each scan print **number of open parenthesis "(" scanned so far – number of close parenthesis ")" scanned so far (please note the minus sign in between)**
- After the sequence is generated it is rearranged randomly.

For example, suppose you have a balanced string = **"( )( )"**
The sequence generated after the first step: **1 0 1 0**
Sequence (one of many) after the rearrangement in the second step: **0 1 0 1**

Your task is - given a sequence in random order, output the lexicographically smallest balanced string if one exists, otherwise output "invalid".

Note: **"("** is lexicographically smaller than **")".** Also remember the string you generate must produce any sequence following above mentioned two steps which can be rearranged to match the sequence given in the input.

Input

There will be **T (1 ≤ T ≤ 11)** test cases. For each test, the first line contains an integer **N (1 ≤ N ≤ 100000)** where **N** denotes the number of integers in the sequence. The second line contains **N** integers separated by spaces describing the sequence. You can assume all the inputs in the sequence will fit in **32** bit signed integer data type.

Output

For each case you have to output the balanced string or "invalid" along with case number. There should not be any intermediate spaces between brackets in the output. See the output format below for more details.

Sample Input	Output for Sample Input
2 4 0 0 1 1 5 1 2 3 4 5	Case 1: ()() Case 2: invalid

F Number of Connected Components

Input: Standard Input
Output: Standard Output

Given **N** nodes, each node is labeled with an integer between **1** and 10^6 **(inclusive and labels are not necessarily distinct)**. Two nodes have an edge between them, if and only if the **GCD** (Greatest Common Divisor) of the labels of these nodes is **greater than 1.** Count the number of connected components in the graph.

Input

First line of the input **T (T * 100)** denotes the number of testcases. Then **T** cases follow. Each case consists of 2 lines. The first line has a number **N (1 * N * 10^5)** denoting the number of nodes. The next line consists of **N** numbers. The **i'th (1 * i * n)** number X_i **(1 * X_i * 10^6)** denotes the label of the node **i**.

Output

For each case you have to print a line consisting consisting the case number followed by an integer which denotes the number of connected components. Look at the output for sample input for details.

Sample Input	Output for Sample Input
2 3 2 3 4 6 2 3 4 5 6 6	Case 1: 2 Case 2: 2

G Extreme XOR Sum

Input: Standard Input
Output: Standard Output

Imagine you have an array of **n** integers **a = [a_0, a_1, a_2, ..., a_{n-1}]**. To find the **extreme sum** of them you have to do the following operations:

1. Create a new list **t = [a_0 + a_1, a_1 + a_2, ..., a_{n-2} + a_{n-1}]**.
2. Let **a = t.**
3. If **a** has only one element remaining, exit. Otherwise go to **1**.

The last remaining element is the extreme sum for the given array. Extreme sum for **a = [1, 2, 4]** is **9**.

To find the extreme XOR Sum, you have to do **XOR operation** instead of addition operation (in the step **1** above).

You are given an array of integers **a**. You have to answer **q** queries. Each query has the form of **b e**. You have to find the extreme XOR sum of the array **[a_b, a_{b+1}, a_{b+2} ... a_e]**.

Input

The first line contains **T (1 ≤ T ≤ 25)**. For each test case:

- The first line contains **n (1 ≤ n ≤ 10^4)**.
- The second line contains **n** integers denoting the array **a**. Each element of the array will be an integer between **0** and **10^9**.
- The third line contains **q (1 ≤ q ≤ 30000)**.
- Each of the next **q** lines contains two integers **b** and **e (0 ≤ b ≤ e < n)**.

Output

For each test case, print the case number in the first line. In the next ¤ lines, print a single line, the extreme XOR sum for the range **[b, e]** for the corresponding query.

Sample Input

```
1
5
1 4 6 7 8
3
0 0
0 1
2 4
```

Output for Sample Input

```
Case 1:
1
5
14
```

H Harmonic Matrix

Input: Standard Input
Output: Standard Output

Matrix is a collection of data, i.e. **[3, 4, 5, 1, 2, 0, 6]** is an example of **1D** matrix of integers. Matrix can be of any dimension. Phase of a matrix is another matrix, defines the comparison of matrix elements with respect to the adjacent elements. Here **3<4, 4<5, 5>1, 1<2, 2>0, 0<6**. If **1** resembles a **'<'** and **0** resembles a **'>'**, phase of the above mentioned matrix is **[1, 1, 0, 1, 0, 1]**.

Now a **2D** matrix can be visualized as a collection of **1D** row matrices placed vertically one after another or as a collection of **1D** column matrices placed horizontally one beside another. The phase of a **2D** matrix is a combination of row phase matrices and column phase matrices. Every single row(/column) of the row(/column) phase matrix is generated from the corresponding row(/column) of the original **2D** matrix. For example, the phase of the **2D** matrix on the left is the combination of two boolean **2D** matrices on the right in the following picture.

83 85 87 15 93 35 84 92 49 21 62 27 90 59 63 26 Original 2D Matrix	1 1 0 0 1 1 0 1 0 0 1 0 Row phase matrix	1 0 0 1 0 0 0 0 1 1 1 0 Column phase matrix

A 2D matrix is harmonic if,

- All the rows of the row phase matrix are same and
- All the columns of the column phase matrix are same.

Following matrix on the left is the shuffled version of the previous one. But it satisfies the above mentioned two criteria. So it's a harmonic matrix.

83 85 87 15 84 92 93 35 21 49 62 26 59 63 90 27 Permuted 2D Matrix	1 1 0 1 1 0 1 1 0 1 1 0 Row phase matrix	1 1 1 1 0 0 0 0 1 1 1 1 Column phase matrix

Given a **2D** matrix of **distinct** integers, your task is to shuffle the elements of the matrix so that it becomes harmonic. For shuffling, you can perform only one kind of operation, take two adjacent elements vertically or horizontally and swap them. You have to sequentially output all the swap operations you need to do to shuffle the given matrix. But the number of swaps you are performing can't be infinite, right?

Input

First line of the input is an integer **T (1 ≤ T ≤ 15)**, the number of test cases. Following lines contain **T** test cases. A case starts with a line containing space separated integers **R** and **C** (**1 ≤ R, C ≤ 1003**) representing the number of row and column of the input matrix. Each of the following **R** lines contains **C** space separated integers which constitutes the input matrix **A**. You can assume that all the elements of matrix A are distinct, strictly positive and do not exceed **1000000009.**

Output

For each test case output contains the test case id in the first line. Next line contains an integer **n**, the number of swap operations you are performing to make the array harmonic. Each of the following **n** lines contains the swap operation formatted with four integers **r1, c1, r2, c2 (1 ≤ r1, r2 ≤ R and 1 ≤ c1, c2 ≤ C)**. Condition **(|r1-r2| + |c1-c2|) = 1** should hold, this means you are swapping two adjacent elements **A(r1, c1)** and **A(r2, c2)**. **The number of swap operations has to be bounded by 2.5*R*C, that means n can be at most 2.5*R*C**. See the sample for exact format. Any valid answer that satisfies the constraints will be accepted.

Sample Input	Output for Sample Input
1 3 3 3 4 6 7 5 8 1 9 2	Case 1: 3 2 1 2 2 3 2 3 3 2 3 3 3

Explanation:

3 4 6 7 5 8 1 9 2 Input Matrix	3 4 6 5 7 8 1 9 2 After swap 2 1 2 2	3 4 6 5 7 8 1 2 9 After swap 3 2 3 3	3 4 6 5 7 9 1 2 8 After swap 2 3 3 3

I In the Kingdom of Hirak

Input: Standard Input
Output: Standard Output

Once upon a time, there was a king named "Hirak Raj". He was so cruel and greedy that farmers in his kingdom had to pay taxes even when they were starving. Diamonds were accumulated in his treasury, but the workers in the mine were not paid properly. Those who tried to protest were taken to "Jantar-mantar Ghar", the chamber for brainwashing people. The king divided his kingdom into **R** regions. People in one region could communicate by sending letters to any other person within the same region but no letter could be sent from one region to another.

A reputed teacher named "Udayan Pandit" was preparing to revolt against the king by educating his disciples in all the regions of the kingdom. Most of his disciples wrote encrypted letters to each other to fix the time and place of their meetings. Any disciple would always forward any letter (s/he wrote or received from others) to other disciples s/he knew in the same region. Udayan Pandit called a collection of his disciples a "group" if a letter written by any member of that group would eventually reach all other members of the group. Moreover, a **group** always contained the maximal number of disciples, i.e. no more disciple could be added to a **group** without violating the property stated above. Hirak Raj knew all the information about who was communicating with whom, but his police could not decipher the encrypted letters and distinguish the secret letters from other normal letters.

Soon Hirak Raj became very desperate to find all the miscreants in the kingdom who want to dethrone him. He ordered every citizen to be taken to an updated version of "Jantar-mantar Ghar", where a device was installed to determine whether a citizen had any intention to revolt or not. All the citizens classified as miscreants were arrested immediately. But the king was not satisfied with the efficiency of the machine. So he made another rule: any citizen who was not yet taken to jail would be arrested if at least **K** members of his/her *group* were already classified as miscreants by the device.

Udayan Pandit noticed that the device in "Jantar-mantar Ghar" is faulty. It classified any person as a miscreant with probability **p** without depending on any information of that person. So, he wanted to know the expected number of arrested disciples in every *region* of the kingdom.

Input

The first line of input file contains a single integer, **T (1 ≤ T ≤ 10)**. Then **T** test cases follow. Each case starts with a line containing two integers, **R (1 ≤ R ≤ 20)** and **K (1 ≤ K ≤ 2000)** and two more integers **a and b (0 < a < b ≤ 1000)**, where **a/b = p**, the probability of classifying a disciple as a miscreant. Then there will be descriptions of the communications in those **R** regions. Each description starts with a line containing two integers, **n (2 ≤ n ≤ 50000)**, number of Udayan Pandit's disciples in the *region* and **m (1 ≤ m ≤ 100000)**, the number of communications between two disciples. Then **m** lines follow. Each of these **m** lines contains two integers **u and v (1 ≤ u, v ≤ n** and **u ≠ v)**, indicating disciple **u** forwards any letter (s/he wrote or received from others) to disciple **v**.

Output

For each case, output "**Case <x>:**" in a separate line, where **x** denotes the case number. Then there will be **R** lines of output for each case. For each of these **R** lines, output "**Region <y>: <z>**", where **y** is the

region number (starting from **1** *to* **R**) and **z** is the expected number of disciples arrested. **Suppose, z = u/v in the reduced form. Print u*(v^-1) mod 1000000007 (10^9 + 7), where v^-1 is the inverse of v mod 1000000007. You may assume that there will be unique modular inverse v^-1 mod 1000000007.**

Sample Input

```
2
2 2 1 2
2 2
2 1
1 2
4 5
1 2
2 3
1 3
3 4
4 2
2 1 1 2
4 5
1 2
2 3
1 3
2 1
3 4
2 2
2 1
1 2
```

Output for Sample Input

```
Case 1:
Region 1: 1
Region 2: 375000005
Case 2:
Region 1: 500000006
Region 2: 500000005
```

Explanation of Region 1 in the 1st test case: There are only two disciples in the region and they form a *group*. Let x be the random variable which denotes the number of disciples arrested. So, E[x] = 1 * Pr(x = 1) + 2 * Pr(x = 2). Now, Pr(x = 1) = Pr(1 was classified as miscreant by the device and 2 was not) + Pr(2 was classified as miscreant by the device and 1 was not) = 0.5*0.5 + 0.5*0.5 = 0.5. For the remaining part, Pr(x = 2) = Pr(Both 1 and 2 were classified as miscreants by the device) + Pr(Either 1 or 2 was classified as miscreant by the device and the other one was arrested later) = 0.5*0.5 + 0 = 0.25. So, E[x] = 1 * 0.5 + 2 * 0.25 = 1, which is the expected number of arrested disciples.

J

Prime Distance

Input: Standard Input
Output: Standard Output

You have an empty **1 * n** grid. The cells of the grid are indexed from **1** to **n** from left to right. You have to put **m** identical coins in the grid. A cell can contain zero or more coins. If you pick a pair of cells each containing at least one coin, the distance between the cells must be a prime number.

How many ways you can place the coins? As the number can be large, find answer modulo 10^9+7. Two ways are different if there is at least one cell which contains different number of coins.

The distance between two cells indexed **i, j** is **|i - j|**.

Input

The first line contains **T (1 ≤ T ≤ 2000)** (the number of test cases). Each of the next **T** lines contains two integers **n (1 ≤ n ≤ 10^5)** and **m(1 ≤ m ≤ 10^5)** separated by a single space.

Output

For each case, print the case number and the answer modulo $10^9 + 7$.

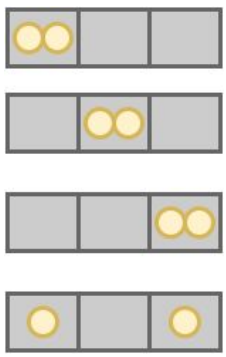
Sample Input

```
2
3 2
6 3
```

Output for Sample Input

```
Case 1: 4
Case 2: 24
```

In the first case, you can put both coins in cell 1, 2 or 3. Or you can put a coin in cell 1 and put another coin in cell 3.



Note that in the 2nd case putting 3 coins in cell 1, 3, 5, is not valid, because the distance between cell 5 and cell 1 is a non-prime.

K	8-ball Rack **Input: Standard Input** **Output: Standard Output**

8-ball is a pool game typically played with **15** object balls. The object balls are of following types:

1. Solids, numbered from **1-7**
2. Stripes, numbered from **9-15**
3. Eight ball: as the name suggests, numbered 8 and is a black solid ball.

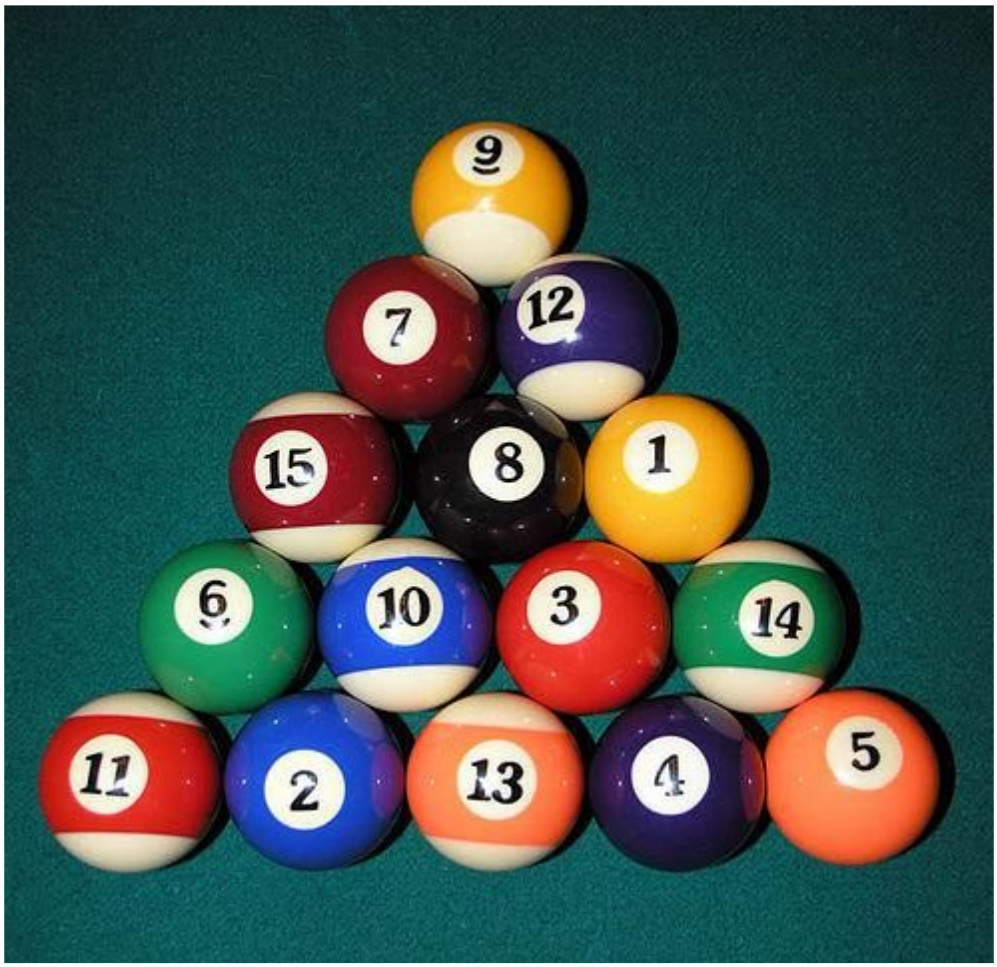


Figure 1: A valid 8 ball rack

In a valid 8-ball rack, the object balls are arranged in a triangle as the figure above, following these rules:

1. The eight ball has to be in the center.
2. The two rear corner balls have to be of different types. If one of them is stripes, then the other one must be from solids. In the picture above, two rear corner balls are 11 and 5. **11** is a stripe and **5** is a solid.

You are given an **8** ball rack (possibly incomplete). If a ball is not present in the rack, then you can assume that they are outside of the rack. You have choice of three different operations:

1. Swap any two balls already in the rack.

2. Pick a ball that's not already in the rack from outside and put that into the rack.
3. Pick a ball that's already in the rack and put it outside.

Given the initial rack and the cost of the different types of operations, find the minimum cost to form a valid 8-ball rack.

Input

The input file starts with a positive integer **T** (**T ≤ 500)** denoting the number of test cases. Each of the **T** test cases starts with three non negative integers, **S**, **P**, and **R**, where,

1. **S** denotes the cost of swapping positions of any two balls.
2. **P** denotes the cost of adding a ball to the rack from outside.
3. **R** denotes the cost of removing a ball from the rack. Note that you can (well, you will have to) return this ball to any place in the rack later.
4. **0 ≤ S, P, R ≤ 100000**

Then come exactly five lines denoting the rack's formation. The **i**'th line has **i** non negative numbers in it. If the **j**'th number on the **i**'th line is **0**, then it means that position is empty. It is guaranteed that no non-zero number appears more than once in the rack.

For more on input specification, please see the sample input section.

Output

For each case, print a single line, **"Case #x: y"**, where x is the case number and y an integer number which denotes the minimum cost to form a valid rack for that case.

Sample Input	Output for Sample Input
2 1 1 1 9 7 12 15 8 1 6 10 3 14 11 2 13 4 5 1 2 3 9 7 12 15 8 1 6 10 3 14 0 0 0 0 0	Case 1: 0 Case 2: 10

Note:

Case 1 is the rack given in the figure 1, which is already valid. So the output is 0

Case 2 is the rack given in the figure 1 only with the difference that, all the balls in the last row is missing. A valid rack can be formed with exactly 5 type 2 operations. As P = 2, the output for case 2 is 2 * 5 = 10.