Computer Science Competition
Region 2025
Programming Problem Set

I. General Notes

1. Do the problems in any order you like. They do not have to be done in order from 1 to 12.

2. All problems have a value of 60 points.

3. There is no extraneous input. All input is exactly as specified in the problem. Unless specified by the problem, integer inputs will not have leading zeros. Unless otherwise specified, your program should read to the end of file.

4. Your program should not print extraneous output. Follow the form exactly as given in the problem.

5. A penalty of 5 points will be assessed each time that an incorrect solution is submitted. This penalty will only be assessed if a solution is ultimately judged as correct.

II. Names of Problems

Number	Name
Problem 1	Arlo
Problem 2	Cole
Problem 3	Elijah
Problem 4	Emerson
Problem 5	Kai
Problem 6	Khan
Problem 7	Melina
Problem 8	Mike
Problem 9	Raymond
Problem 10	Remy
Problem 11	Saim
Problem 12	Veerasamy

1. Arlo

Program Name: Arlo.java **Input File: none**

Arlo is drilling to try out for the yell squad at his school. He needs to chant the phrase “Let's go Tyrannosauruses!” exactly 15 times.

Input: none

Output: Print the phrase “Let's go Tyrannosauruses!” 15 times.

Sample input:
```
None
```

Sample output:
```
Let's go Tyrannosauruses!
Let's go Tyrannosauruses!
Let's go Tyrannosauruses!
Let's go Tyrannosauruses!
Let's go Tyrannosauruses!
Let's go Tyrannosauruses!
Let's go Tyrannosauruses!
Let's go Tyrannosauruses!
Let's go Tyrannosauruses!
Let's go Tyrannosauruses!
Let's go Tyrannosauruses!
Let's go Tyrannosauruses!
Let's go Tyrannosauruses!
Let's go Tyrannosauruses!
Let's go Tyrannosauruses!
```

2. Cole

Program Name: Cole.java **Input File: cole.dat**

Cole is revisiting his old favorite game, *Factorio*, now with a new DLC. A core mechanic of the game is automation — namely, anything that can be crafted can also be automated using *Assembling Machines*. These machines take inputs, process recipes, and produce outputs, which can then be used in more advanced recipes.

Previously, Cole relied on online guides to determine the correct input-output ratios. Now, he's much smarter, and wants to calculate them himself. Given a set of crafting recipes and a target production rate for a specific item, determine how many assembling machines are needed for each input.

Input: The first line will consist of two space-separated integers R and Q, ($1 \leq R \leq 300$, $1 \leq Q \leq 10^4$), denoting the number of crafting recipes and queries. The next R lines will all take the same format:

$$S, p/t, n, (i_1, q_1), (i_2, q_2), \ldots, (i_n, q_n)$$

Where…

- S is the item name.
- p is the number of items produced per crafting cycle.
- t is the time taken per crafting cycle.
- n is the number of inputs required.
- Each pair (i_k, q_k) represents an input item and its required quantity.

Raw ingredients (available infinitely, but take time to harvest) have $n = 0$, $p = 1$, and $t = 1$, will be written as:

$$S, 1/1, 0$$

Every input item in a recipe is guaranteed to have a corresponding entry among the R recipes. Moreover, there are no cycles in the dependencies, and there will be *at least* one crafting recipe that is a raw ingredient.

The next Q lines contain queries in the format "A, r" where A is the item to be produced, and r ($10^{-4} < r \leq 10^6$) is the desired output rate per second (up to 5 decimal places). A is guaranteed to be in the recipes.

Output: For each of Cole's Q queries…

1. Print "A (r):", where r is formatted to 5 decimal places.
2. For each required input (i_k), print a tab followed by "i_k: a_k", where a_k is the number of assembling machines needed, rounded to the nearest whole number. List inputs in lexicographical order.

Note that we are assuming that r is an average rate that needs to be met over an effectively infinite amount of time and need not be satisfied at rate r immediately.

Hint: To avoid floating-point precision errors when rounding, subtract a small value $\varepsilon = 10^{-6}$ before rounding.

~ ***Sample input and output on next page*** ~

~ *Cole continued* ~

Sample input: (*indented lines are a continuation of the previous line*)

```
10 3
ExpressTransportBelt, 1/0.5, 3, (FastTransportBelt, 1), (IronGearWheel, 10),
      (Lubricant, 20)
FastTransportBelt, 1/0.5, 2, (TransportBelt, 1), (IronGearWheel, 5)
TransportBelt, 2/0.5, 2, (IronGearWheel, 1), (IronPlate, 1)
IronGearWheel, 1/0.5, 1, (IronPlate, 2)
IronPlate, 1/3.2, 1, (IronOre, 1)
Lubricant, 10/1, 1, (HeavyOil, 10)
HeavyOil, 25/5, 2, (CrudeOil, 100), (Water, 50)
IronOre, 1/1, 0
CrudeOil, 1/1, 0
Water, 1/1, 0
TransportBelt, 20
FastTransportBelt, 10
ExpressTransportBelt, 5
```

Sample output:

```
TransportBelt (20.00000/s):
      IronGearWheel: 5
      IronOre: 30
      IronPlate: 96
      TransportBelt: 5
FastTransportBelt (10.00000/s):
      FastTransportBelt: 5
      IronGearWheel: 28
      IronOre: 115
      IronPlate: 368
      TransportBelt: 3
ExpressTransportBelt (5.00000/s):
      CrudeOil: 400
      ExpressTransportBelt: 3
      FastTransportBelt: 3
      HeavyOil: 20
      IronGearWheel: 39
      IronOre: 158
      IronPlate: 504
      Lubricant: 10
      TransportBelt: 2
      Water: 200
```

3. Elijah

Program Name: Elijah.java **Input File: elijah.dat**

Elijah is working on writing a program that rotates images, but he's having trouble figuring it out. He knows you're quite apt at programming and has enlisted you to help him out.

Input: The first input line will contain a single number N (1 <= N <= 100) denoting the number of inputs to follow. Each input will start with two values R, C (1 <= R, C <= 100) denoting the number of rows and columns respectively of the image to follow. The following R lines will contain C characters, representing the image Elijah is supposed to rotate. The final line of each input will be a single value V (1 <= V <= 10^20) denoting the number of degrees to rotate the image.

Note: V is guaranteed to be a multiple of 90, and all rotations are clockwise.

Output: Output each image rotated with a blank line separating each image.

Sample input:

```
1
11 10
@@@@@@@@@@
@        @
@  ****  @
@  *  *  @
@  *  *  @
@  *  *  @
@  *  *  @
@  *  *  @
@  ****  @
@        @
@@@@@@@@@@
90
```

Sample output:

```
@@@@@@@@@@
@        @
@        @
@ ******* @
@ *      * @
@ *      * @
@ ******* @
@        @
@        @
@@@@@@@@@@
```

4. Emerson

Program Name: Emerson.java **Input File: emerson.dat**

Emerson has designed a new video game for you to test. It's a maze solving game, with a portal gun. You need to determine the shortest path to escape each level of the maze, if you use the portal gun optimally. The portal gun will have a specified number of charges, and each charge will make a portal from your current position, to any position that is 2 spaces away, or 1 diagonally (2 up, 1 up 1 left, 2 left, 1 left 1 down, 2 down, 1 down 1 right, 2 right, 1 right 1 up), and using a portal will not count as a step. When not using a portal, you can only move in the 4 cardinal directions (up, down, left, right). The maze will be made up of empty paths, walls, and land-sharks which you cannot get within 2 spaces in front of whichever direction they are facing (including when entering or exiting a portal). The start and end points of the maze will never be within the restricted area of a land shark.

Input: The input will begin with an integer, `n` (`0 < n <= 1000`), denoting the number of test cases to follow. Each test case will consist of three integers, separated by a space, `r`, `c`, and `p` (`0 < r,c <= 250 0 < p < 13`), denoting the number of rows and columns in the maze level, and the number of "charges" the portal gun has. The following r lines will each contain c characters denoting the layout of the maze level, which will be made up of the following:

- `.` – Denotes an empty space in the level, where you can walk.
- `#` – Denotes a wall in the level, an impassable object.
- `S` – Denotes the starting point for the level.
- `E` – Denotes the end point of the level.
- `< > ^ v` – Any of the preceding characters will denote a land shark, with the direction their "mouth" is pointed being the direction they are facing (< is right, > is left, `^` is down, `v` is up). Land sharks do not move throughout the maze, they are rather lazy.

Output: For each test case, output the minimum amount of steps required to get from the starting point to the ending point of the maze. If this is not possible, output `-1`.

Sample input:

```
2
6 7 2
S..<...
..#..vE
#..##..
^....##
..>..#.
....v..
8 4 1
E..>
..#.
.#.v
##..
<...
.#.#
S..#
####
```

Sample output:

```
3
-1
```

5. Kai

Program Name: Kai.java **Input File: kai.dat**

It's springtime and Kai hears the familiar sound of baby chicks coming from his family's chicken coop in the back yard. Kai decides it's time for a parade of the new chicks. Display the chicks in a row based on the number of chicks in the data file.

Input: The input will consist of a single integer N (1 <= N <= 10) representing the number of chicks to display in the parade.

Output: The given parade of chicks in the form as shown below.

Sample input:

```
3
```

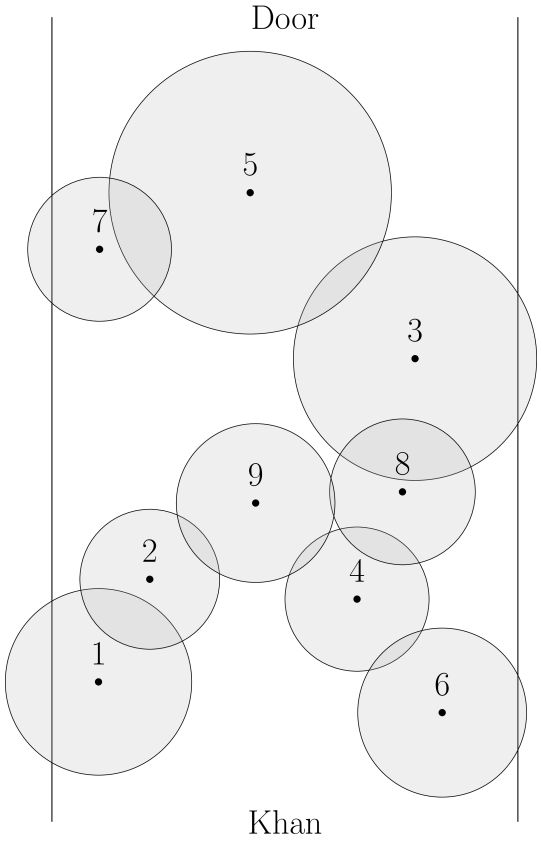
Sample output:

```
 _   _   _
(*> (*> (*>
//\ //\ //\
V_/_V_/_V_/_
```

6. Khan

Program Name: Khan.java **Input File: khan.dat**

Khan is the original person that the "House Robber" problem (also known as the "Knapsack" problem) was written about. What often doesn't get talked about with the house robber problem is that after successfully optimizing which items Khan places in his knapsack, he now must escape the house he has just robbed. However, in his desire to have perfectly optimized the items in his bag, he has taken too long and has triggered the alarm of the house! However, thankfully for Khan's sake, the homeowner is using an experimental alarm which is comprised of a series of sensors that incrementally turn on. That is, at time $t = 1$, sensor 1 turns on and can detect any moving entity within its detection radius. At time $t = 2$, sensor 2 can do the same, and so on. Khan has managed to make his way to the final hallway; however, this is the most complex of them all so far. Help Khan determine how much time he has left before he will be unable to navigate out of the final hallway and make it to the door without being detected.



Input: The first line of input will consist of a single integer, T ($1 \leq T \leq 10^2$), denoting the number of testcases to follow. Each testcase will begin with two space-separated integers n_i ($1 \leq n_i \leq 10^4$) and w_i ($1 \leq w_i \leq 2^{16}$), denoting the number of sensors and the width of the hall for the i^{th} testcase, respectively. The following n_i lines will each contain three space-separated floating points x_j, y_j, r_j, the j^{th} of which denotes the (x_j, y_j) center and effective radius of the j^{th} sensor. You may assume that $0 \leq x_j \leq w_i$, $0 \leq y_j \leq 10 \cdot w_i$, and $0 < r_j \leq w_i$ for all j.

Output: For each of Khan's T escape situations, on its own line, print out a single integer t_i, denoting the maximum amount of time that Khan has before he is unable to escape undetected. If Khan is free to leave undetected after all n sensors turn on, instead print "`Completely Undetected.`". You may assume that Khan may be represented as a point and, as a result, if two circles are tangential to one another, their single point of intersection blocks Khan. Moreover, assume the hall occupies the space starting from $x = 0$ to $x = w_i$. If it is impossible for Khan to escape if the first sensor turns on, print `0` indicating that Khan should have left sooner. Lastly, for your answer to be considered correct, ensure that floating point comparisons are accurate within an absolute error/precision of $\varepsilon = 10^{-6}$.

~ ***Sample input and output on next page*** ~

~ ***Khan continued*** ~

Sample input:

```
5
9 10
1.0 3.0 2.0
2.1 5.2 1.5
7.793 9.93 2.611
6.548 4.774 1.546
4.255 13.488 3.031
8.375 2.341 1.812
1.023 12.274 1.543
7.523 7.074 1.563
4.372 6.834 1.702
6 15
2.7 17.1 4.35
12.3 16.8 4.35
6.6 12.75 3.15
6.975 7.875 3.15
12.525 6.375 4.35
2.1 3.3 4.35
6 25
4.5 28.5 7.25
3.5 5.5 7.25
20.5 28 7.25
11 21.25 5.25
11.625 13.125 5.25
20.875 10.625 7.25
6 13
1.82 2.86 3.77
2.34 14.82 3.77
5.72 11.05 2.73
6.045 6.825 2.73
10.66 14.56 3.77
10.855 5.525 3.77
3 7
3.5 5.25 3.535
1.05 1.05 0.35
5.95 1.05 3.5
```

Sample output:

```
6
2
3
4
0
```

7. Melina

Program Name: Melina.java **Input File: melina.dat**

You have made a new friend, Melina. She has trouble counting sometimes, but she recently accepted a job as a cashier at a taco shop. You need to write a program to determine how many different ways she can make the change required by the purchase each customer makes, given the money denominations she has in the cash register.

Input: The input will begin with an integer, `n` (`0 < n <= 1000`), denoting the number of test cases to follow. Each test case will consist of two lines. The first will contain two space-separated floating-point numbers, `a` and `t`, denoting the amount the customer paid, and the amount that was due. The following line will contain an unspecified number of space-separated integers denoting all the possible denominations of money Melina can use to make change, it can be assumed for our purpose that you have access to an infinite number of each denomination. Denominations will never be smaller than `.01` (1 cent), and neither `a` or `t` will exceed two decimal places. Additionally, `a` is guaranteed to be larger than `t`, and the difference between the two will never exceed `10^9`.

Output: For each test case, output the number of possible combinations Melina can make to make the amount of change required by the purchase each customer made.

Sample input:

```
3
25 24.78
.01 .05 .10 .25 1
40 39.15
.01 .05 .10 .25 1
50.26 41.95
.01 .05 .10 .25 1 2 5 10
```

Sample output:

```
9
163
332365
```

8. Mike

Program Name: Mike.java **Input File: mike.dat**

You've been volunteering at the local animal shelter in your free time, in particular, you've been put in charge of the cats. Your boss, Mike, has asked you to put in the order for cat food this month. When you asked him how that's typically done he said that the shelter just orders some massive amount of each food, and rotates through them every day. You, however, have noticed that some cats like certain flavors of cat food and won't even eat others. You've gone through the liberty of finding out which flavors the cats like to eat when they eat, and how much they eat at a time. You've asked your boss if you can order based on this data and he says sure but only if you spend as little money as possible.

Input: The first input line will contain a single value N ($1 \leq N \leq 100$) denoting the number of test cases to follow. Each test case will start with a value M ($1 \leq M \leq 1000$) that denotes the number of cat feedings that follow. The next M lines will be in the form of T, A, and B ($1 \leq A \leq 1000$). T is the hour, minute, and second that this cat will be fed. A is the amount of food in ounces that the cat eats at time T. B is the brand of food that the cat is fed at time T. Each test case will end by enumerating all brands B required to feed all cats. Each brand will have an input line that starts with the brand name and then a list of purchasable cans of food. The list of purchasable food will take the form of C:W:P where C ($1 \leq C \leq 100$) is the number of cans in this package, W ($1 \leq W \leq 1000$) is the weight of each can of food in ounces, and P is the price of this package rounded to the nearest cent.

W is guaranteed to be an integer, and every brand will at least have single one-ounce cans for sale.

Output: Output "`Total cost to feed all cats: `" followed by the least amount of money you need to spend to feed all the cats their desired foods, rounded to the nearest cent with commas where required.

Note that you cannot have less food than required to feed all cats, but you may have extra.

Sample input:

```
1
10
20:43:13 3.55 super-premium
17:48:10 0.01 premium
13:40:36 4.08 premium
12:11:33 0.58 normal
06:27:30 2.62 super-premium
02:38:30 0.76 premium
12:14:50 1.96 super-premium
11:49:22 3.51 normal
08:42:20 0.30 generic
02:59:08 4.38 generic
normal 1:1:1.14 6:5:15.25 12:3:19.67
generic 1:1:0.98 6:3:12.52 24:5:33.17
premium 1:1:1.99 6:5:25.99 12:3:39.99
super-premium 1:1:5.99 6:5:79.99 12:12:212.12
```

Sample output:

```
Total cost to feed all cats: $74.46
```

9. Raymond

Program Name: Raymond.java **Input File: raymond.dat**

Your classmate Raymond is a bit of an idiot. You've explained to him countless times what LCM (Lowest Common Multiple) means and how to find it between two numbers but he just doesn't get it. Feeling sorry for him, you've decided to go ahead and just write a program that given a list of numbers will let Raymond know the LCM.

Input: The first input line will contain a value N ($1 \leq N \leq 100$) denoting the number of test cases. Each test case will start with a single value M ($1 \leq M \leq 100$) representing the number of values in the list. The next line will contain M integers A such that ($1 \leq A \leq 1000$), of which you are to find the LCM between them all.

Output: Output "`Lowest Common Multiple is` **x**" where **x** is the lowest common multiple. If the LCM is 1, then instead print "`LCM NUMBER 1!`" to give Raymond a little excitement.

Sample input:

```
2
5
1 1 1 1 1
3
159 261 932
```

Sample output:

```
LCM NUMBER 1!
Lowest Common Multiple is 12892356
```

10. Remy

Program Name: Remy.java **Input File: remy.dat**

The IT department of the store where Remy is working has been having some issues with their website recently. Remy has been tasked with analyzing their log file to gather some data. There are five (5) items to report: the total number of requests listed in the log file, the number of unique IP addresses that made requests, the IP address that made the most requests and the number of requests it made, the percentage of requests that resulted in an error status code (4xx or 5xx), and the most requested URL and the number of times it was requested.

Input: The data file consists of an unspecified number of lines in the following format:

```
[Timestamp] IP_Address HTTP_Method URL Status_Code
```

- **Timestamp:** The date and time of the request in the format `YYYY-MM-DD HH:MM:SS`.
- **IP_Address:** The IP address of the client making the request.
- **HTTP_Method:** The HTTP method used (e.g., `GET`, `POST`, `PUT`, `DELETE`).
- **URL:** The URL that was requested.
- **Status_Code:** The HTTP status code returned by the server (e.g., `200` for OK, `404` for Not Found, `500` for Server Error).

Output: see sample output below for required wording and formatting

- Print the total number of requests.
- Print the number of unique IP addresses.
- Print the most frequent IP address and the number of requests it made.
- Print the error rate as a percentage, rounded to two decimal places.
- Print the most requested URL and the number of requests it received.

Sample input:

```
[2024-10-27 10:00:00] 192.168.1.1 GET /index.html 200
[2024-10-27 10:00:05] 192.168.1.2 POST /login 200
[2024-10-27 10:00:10] 192.168.1.1 GET /index.html 200
[2024-10-27 10:00:15] 192.168.1.3 GET /products 404
[2024-10-27 10:00:20] 192.168.1.1 GET /about 200
[2024-10-27 10:00:25] 192.168.1.4 POST /submit 500
[2024-10-27 10:00:30] 192.168.1.2 GET /index.html 200
```

Sample output:

```
Total requests: 7
Unique IP addresses: 4
Most frequent IP address: 192.168.1.1 (3 requests)
Error rate: 28.57%
Most requested URL: /index.html (3 requests)
```

11. Saim

Program Name: Saim.java **Input File: saim.dat**

Saim is a new kid in your class and you've decided you're going to be his friend. He really likes the video game Mortal Kombat. You have built your own version of this game called Kortal Mombat to play with him. You need to write a program to read in the commands given and determine who wins each game. The game is similar to most fighting games, where two characters face off, doing different actions until one character runs out of health. You will go first, doing one action, then Saim will go (you get advantage because Saim is an experienced player). Kortal Mombat has 4 different character options who do different damage and healing, and each character will have 4 possible actions they can take, as follows:

- `RT` – This is the character's primary attack, it will do a certain amount of damage, depending on the character.
- `LT` – This is the block, each character has the same block. This will negate the damage of the next attack by the opponent, but will do nothing if the next move by the opponent is not an attack.
- `RB` – This will be a secondary attack, typically does more damage, but you will take double damage from the opponent's next attack.
- `LB` – This will be a heal, which will add a small amount of health back to your character.

The 4 characters are as follows:

Character	`RT` Damage	`RB` Damage	`LB` Health Gained	Total Health
A	100	200	40	1000
B	100	260	20	1500
C	120	220	50	800
D	80	150	80	1000

Input: The input will begin with an integer, `n` (`0 < n <= 1000`), denoting the number of test cases to follow. Each test case will begin with a line containing 2 chars denoting the characters chosen by you and Saim respectively, followed by an integer `m` denoting the number of commands each of you input. The following line will contain m commands (from the 4 above actions you can take), denoting the actions you took in the game. The line after will contain the same information for Saim. If one character's health goes to 0 before the end of the line, they lose, and you should ignore the rest of the line. When `m=0`, there will be no following input lines for that case.

Output: For each test case, output the name of the winner, either "`You`" or "`Saim`". If you get to the end of both player's actions and neither has won, output "`Draw`". After all test cases, output your record, in the format `W-D-L`, where `W` is the number of your wins, `D` is the number of your draws, and `L` is the number of your losses.

Sample input:

```
2
A B 9
RT RT RT RB LB RT RT RT RT
RB RT RB RT RB RT RB RB RT
A C 9
RB RB RT RT RB RB RT RT RB
RB LB RB LB RB LB RB LB RB
```

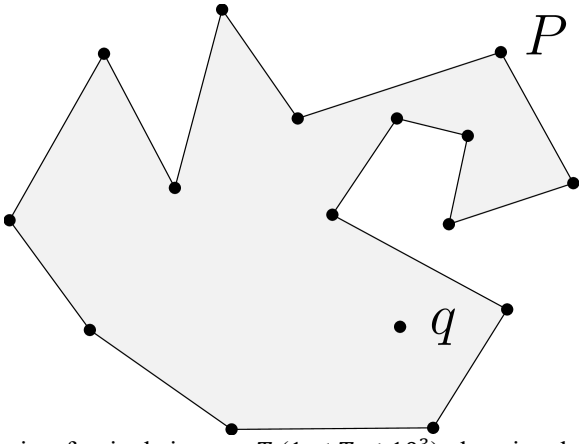
Sample output:

```
Saim
You
1-0-1
```

12. Veerasamy

Program Name: Veerasamy.java **Input File: veerasamy.dat**

Veerasamy has recently found himself confined to two dimensions and is very confused by this planar projection. He has several questions related to his recent transformation, including why he is now only being represented as a single point, but he is really curious in knowing whether he is still inside the "room" he was in back when he was in three-dimensional space. Help Veerasamy among his confusion determine if he is still contained within the, now two-dimensional, representation of the room he was in.



Input: The first line of input will consist of a single integer, T ($1 \leq T \leq 10^3$), denoting the number of testcases to follow. Each testcase will begin with an integer n ($3 \leq n \leq 10^3$) denoting the number of points that comprise the simple polygon P followed by a space and either the word `"Convex"` or `"Concave"` indicating whether the polygon is convex or concave. The next $n + 1$ lines will consist of two space-separated floating-point numbers x_i and y_i, ($-10^4 \leq x_i, y_i \leq 10^4$) denoting the x- and y-coordinate of the i^{th} point (p_i), the last of which denotes the point q (Veerasamy's location), which is not a part of the polygon. The polygon is formed by creating an edge between p_i and $p_{(i+1) \bmod n}$ for all $1 \leq i \leq n$. It should be noted that the points of P will be listed starting from the left-most point, proceeding along the incident edge that is clockwise from the first point, continuing along the unvisited incident edge of p_i, until p_n (which has an edge with p_1). Lastly, general position can be assumed (no two points share the same x-coordinate / y-coordinate; no three points are collinear).

Output: For each of Veerasamy's T situations, on its own line, either print `"Safe and sound."` if Veerasamy is still in his room (inside of polygon P), or print `"Dreaded dimensional downgrade!"` if he no longer is in his room (outside of polygon P).

Sample input:

```
3
15 Concave
1.44 2.65
2.64 4.76
3.54 3.06
4.14 5.32
5.10 3.94
7.68 4.78
8.60 3.12
7.02 2.6
7.26 3.72
6.36 3.94
5.54 2.72
7.76 1.52
6.82 0.02
4.26 0.00
2.46 1.26
6.40 1.30
```

```
9 Convex
1.44 2.65
2.64 4.76
4.14 5.32
7.68 4.78
8.60 3.12
7.76 1.52
6.82 0.02
4.26 0.00
2.46 1.26
6.40 1.30
```

```
9 Convex
1.44 2.65
2.64 4.76
4.14 5.32
7.68 4.78
8.60 3.12
7.76 1.52
6.82 0.02
4.26 0.00
2.46 1.26
8.25 5.00
```

Sample output:

```
Safe and sound.
Safe and sound.
Dreaded dimensional downgrade!
```