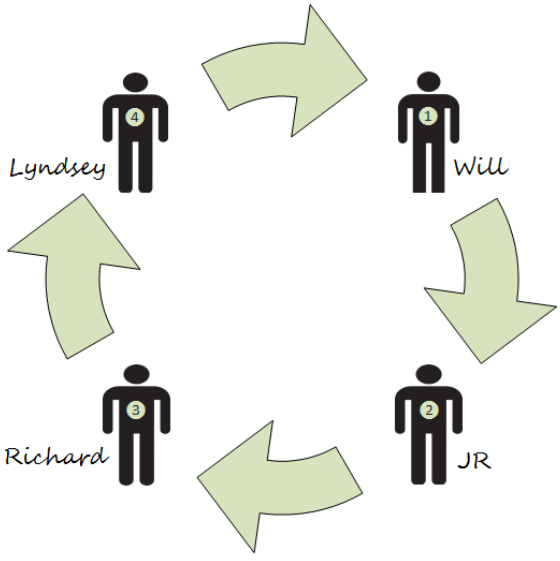
1. Whoops!

Back before there were fancy telephones with highly sophisticated games like Candy Crush and Angry Birds, people had to come up with ways of entertaining themselves. One very simple, yet challenging and entertaining game is called, 'Whoops!'

Whoops! can be played with 3 or more players. The players are arranged in a 'circle' to establish an order. The starting player says 1, and the following player says 2, and so on. It always starts in a clockwise rotation. This continues until a number is reached that meets one of the following three qualifications:
The number contains a 7 (e.g. 17)
The number is evenly divisible by 7 (e.g. 35)
The sum of the digits is equal to 7 (e.g. 52, as 5 + 2 = 7)
When one of those numbers is reached, rather than saying the number, the person says 'Whoops!' Then two things happen. That number is skipped, **AND** the order of the game proceeds in the opposite direction. Anyone can start the first round. Each subsequent round is started by the loser of the previous round. The loser is whoever said the wrong number, went out of turn, or forgot to go.

Example:
Four players, Will, JR, Richard, and Lyndsey are in a circle. They are in that order. Will starts, and, as all games do, it starts in clockwise order.

Will-1
JR-2
Richard-3
Lyndsey-4
Will: 5
JR: 6
Richard: Whoops! *(reverse direction)*
JR: 8
Will: 9
Lyndsey-10
Richard-11
JR-12
Will-13
Lyndsey-Whoops! *(reverse direction)*
Will-15
and so on



Design a program to play Whoops! Below I list six potential players, with their flaws. Your program must account for them. It will read from standard in and write results to standard out. The input will consist of a space separated list of the players, in the order that they will start the first game. This will be followed by the number of rounds this group of players will play. This will repeat until no input is available.

The output should consist of the player's name, followed by a dash, followed by their response. If their response is not correct, the next line should say simply END OF ROUND with no punctuation. After the last round, starting with the first player, print out each player's name, followed by a : followed by a space, and then the number of rounds that player lost *during that game*. Then print "--" to separate different games, or to indicate the end of the file. *(the last character output SHOULD be a new line)*

Players

Richard forgets to go and says nothing immediately after a direction change (a Whoops!)
Michael forgets rule number 1, to check each digit for 7
Tim forgets rule 2, numbers evenly divisible by 7
Lyndsey forgets rule 3, digits add up to 7
JR ignores all three rules. On his turn, he just says the next number. He never says Whoops!
All other players (e.g. **Will**, **Brigid**, etc) should be considered perfect, in that they never make a mistake.
Note: Forgetting rules means he/she will say the number instead of Whoops! and lose the round. Forgetting to go means he/she doesn't say anything, and therefore loses the round.

Sample Input

```
Lyndsey JR Richard
3
Brigid Will Michael
1
```

Sample Output

```
Lyndsey-1
JR-2
Richard-3
Lyndsey-4
JR-5
Richard-6
Lyndsey-Whoops!
Richard-
END OF ROUND
Richard-1
Lyndsey-2
JR-3
Richard-4
Lyndsey-5
JR-6
Richard-Whoops!
JR-8
Lyndsey-9
Richard-10
JR-11
Lyndsey-12
Richard-13
JR-14
END OF ROUND
```

JR-1
Richard-2
Lyndsey-3
JR-4
Richard-5
Lyndsey-6
JR-7
END OF ROUND
Lyndsey: 0
JR: 2
Richard: 1
--
Brigid-1
Will-2
Tim-3
Brigid-4
Will-5
Tim-6
Brigid-Whoops!
Tim-8
Will-9
Brigid-10
Tim-11
Will-12
Brigid-13
Tim-14
END OF ROUND
Brigid: 0
Will: 0
Tim: 1
--

2. 2D Island Pools

Imagine a 2 dimensional island in the ocean, with a topography defined by an array of numbers. Each array index represents a meter in length of land, and the number represents the height in meters, with 0 representing sea level.

Ex: array[5] = { 0, 2, 1, 4, 0 }

This example would roughly look something like this (with X indicating land):

```
  X
  X
 X X
 XXX
XXXXX
```

Now imagine there is a torrential rain that occurs on the island. Water will pool in any valleys created in the 2 dimensional island. Water will flow downhill and into the ocean if there is no valley to contain it. In the above example, water would be contained for 1 square meter in the 3rd column due to the valley, but all other water will be able to flow sideways and down into the ocean on either side.

The island would then look like this (with X indicating land, and O indicating pooled water):

```
  X
  X
 XOX
 XXX
XXXXX
```

You will be given inputs to describe various 2 dimensional islands and required to compute the square meters of pooled water that will remain after a torrential rain. Assume there is always enough rain to fill in all valleys.

Each line of input describes 1 island, with a valid input line containing only integers to describe the island. The first number will be the the array size, and the remaining numbers on the line will be the array entries. A line with only the number 0 indicates the end of input. For each island, output should be the number of square meters of water that will pool on the island, or the phrase "Invalid input" if the input line is invalid.

Sample Input:

```
5 0 3 1 4 0
3 0 1.5 0
4 2 2 2 2
5 1 2 3 2 1
5 3 2 0 2 3
0
```

Sample Output:

```
2
Invalid input
0
0
5
```

3. Arithmetic and Geometric Progressions

A math teacher has written an excellent paper for his students about arithmetic and geometric progression. Since it was so well received, many others want to use this paper, but using different series, due to cultural preferences. Furthermore the text will be translated to different languages, which the math teacher does not speak. On the other hand he must validate the correctness of the actual used series.

Problem Statement

For this he wants have a program which scans the text, finds all the series, determine whether it is arithmetic or a geometric one and verify whether all show numbers are correct.

For the input you may assume that the only numbers in the text are from an arithmetic or geometric series. All numbers are non-negative integrals, without a sign. Also the first 3 numbers of the series are always correct. Subsequent numbers in a series are separated by spaces, words and/or commas. The last number of a series is optionally followed by some words and eventually a period (full stop?). Note that a number may get too large to fit on a line, so then the rest of the number is on the next line. The languages used in the papers is simple text using 8 bit ASCII, so no Unicode or any HTML tags or other text formatting options.

The output must be: a sequence number for the found series, a dot, the start line of this series, a colon, indication of arithmetic or geometric progression, a colon, the length of the series and the wrong terms, separated by commas, each with the given value, a colon and their correct value. At the end the last line must hold the number of found arithmetic and geometric series, as well as the total number of errors.

Sample Input

The simplest example of arithmetic progress is 1, 2, 3, 4, 5 and so on. A well-known geometric progression is: 1, 2, 4, 8, 16, 32, 64... Some arithmetic progress may grow fast, like 1, 1001, 2001, 3000, 4001. Geometric progression may start slow, but in the end they will win, for example: 2, 6, 18, 54, 162, 486, 1458, 4474, 13122, 39366, 118098, 354294, 1062882, 3188646, 9565938, 28679814, 86093442, 25 8280326 and then 774840978 gets you in less than twenty steps close to a billion.

Sample Output

1. Line 1: arithmetic: 5 terms, 0 errors
2. Line 2: geometric: 7 terms, 0 errors
3. Line 2: arithmetic: 5 terms, 1 error: 3000: 3001
4. Line 3: geometric: 19 terms, 2 errors: 4474: 4374, 28679814: 28697814

Total of 2 arithmetic and 2 geometric progressions, total of 3 errors.

4. Crashing Robots

For this problem it is your job to determine if a robot can be guided around a grid from its start point to its goal. However, because of a minor mechanical defect the robots have no brakes. This means when a robot is propelled in one direction it continues in that direction until it crashes into a wall.

Details

For input you will be given the description of a square grid, walls, a starting square and goal square. You must determine for a given starting point if the robot can reach the goal by only moving up, down, left or right. Remember once sent in a direction a robot will only stop when it hits a wall. The starting point and goal will both be given as integers representing grid squares. All grids are numbered left to right starting with 0 in the top left. Walls will be described for each square as a binary string where 1 represents a wall and 0 means no wall. The position in the binary string represents where in the square the wall is starting at the top and moving clockwise.

Notes

The grid will always be square.
The robot must be stopped on the goal not just pass over it.

Example Grid

This input
4
3
1001 1000 1000 1100 0101 0001 0000 0100 0001 0000 0000 0100 0011 0010 0010 0110

Defines the grid

0	1	2	3 [G]
4 [S]	5	6	7
8	9	10	11
12	13	14	15

Square 4 is the start.
Square 3 is the goal.
The entire outside of the grid has walls.
Square 4 and 5 have a wall between them.

Input & Output

Each test case will be on three lines on standard in. The first line will be the start position. The second line will be the goal. The final line will be the board description. All lines should be read until an EOF. For each test case on standard in you need to print to standard out if the goal can or can not be reached. If the goal is reachable return “Path found.” when the goal is not reachable return “Path not found.”.

Sample Input

```
4
3
1001 1000 1000 1100 0101 0001 0000 0100 0001 0000 0000 0100 0011 0010 0010 0110
4
7
1001 1000 1000 1100 0101 0001 0000 0100 0001 0000 0000 0100 0011 0010 0010 0110
```

Sample Output

```
Path found.
Path not found.
```

5. Look and Say

A look-and-say sequence is generated by reading the digits of the previous member, and counting the number of integers in groups of the same integer. The next member is generated by placing the counts for each group of integers in front of the integer. For example, starting with the integer 1, the look-and-say sequence is as follows:

1, 11, 21, 1211, 111221, 312211, 13112221, 1113213211, ⋯

To generate this sequence:
1 is read off as "one 1" (or "1 1"), and so it produces 11
11 is read off as "two 1s" (or "2 1"), and so it produces 21
21 is read off as "one 2, then one 1" (or "1 2, 1 1"), and so it produces 121
1211 is read off as "one 1, then one 2, then two 1s" (or "1 1, 1 2, 2 1"), and so it produces 111221
111221 is read off as "three 1s, then two 2s, then one 1" (or "3 1, 2 2, 1 1"), and so it produces 312211
⋯ and so on.

Problem Statement

You will be given two inputs separated by a space. The first input is the string of integers that makes up the starting member of the sequence. The second input is the n^{th} member to return. If the input is 0 0, you are finished. For instance, if the input is

12 5

then the starting member is 12. Assuming this is the 0^{th} member, the next 5 members of the sequence would be

1112
3112
132112
1113122112
311311222112

You would submit 311311222112 as your answer.

Sample Input

```
12 5
313 10
8 8
11111111111111111111 9
0 0
```

Sample Output

```
311311222112
1311121312211213211312111322211213211321322123211211131211121332211 3
1113122113121113222118
1113122113121113222112111312211312111322211031131211131221
```

Assumptions

All numbers are integers 0 through 9
In the initial member, no grouping of same integers will be longer that 99 digits
The length of the initial member will be between 1 and 300 digits (inclusive)
The maximum iteration of the sequence you will be asked to produce is the 30^{th} member

More Examples

2 9 =>
12
1112
3112
132112
1113122112
311311222112
13211321322112
1113122113121113222112
3113112221131112311332211 2

1776 8 =>
112716
2112171116
12211211173116
112221123117132116
213221121321171113122116
121113222112111312211731131122211 6
111231133221123113112221171321132211 6
311213212322211213211321322117111312211312111322211 6

112222255 6 =>
215225
1211152215
11123115221115
3112132115223115
132112111312211522132115
111312211231131122211522111312 2115

22222222222 5 =>
112
2112
122112
11222112
21322112

6. Space Battle Simulation

You are a part of a galactic empire that is at war. Your spies are able to determine the exact configuration of the enemy fleet before battle is joined. (The enemy is also rather fixed in their thinking and will not change their formation before or during the battle.) With this intelligence the supreme commander has asked you to devise a program that will tell him if his ships will be able to defeat the enemy fleet before battle is joined. The commander has specified that in order for him to consider it a victory none of the enemy ships may remain and his fleet should contain at least one ship. If you determine that your fleet cannot win your program should print out "RETREAT". If it is possible to defeat the opposing fleet your program should print "VICTORY IS ASSURED".

During battle your first ship will engage the enemy's first ship until one or both are destroyed. The next ship from the opposing fleet will engage until one or both are destroyed. If both are destroyed then the next ship from both fleets will engage. This process will repeat until the battle is decided. An engagement consists of a series of rounds. Each round consists of applying each ship's base damage plus any special abilities against the opposing ship's health value (HP). An engagement is ended when one or both ships have been destroyed. For consistency assume that per round both ships attack each other simultaneously. Thus even though an undamaged Fighter (F) will instantly destroy a Bomber it will still take 200 damage.

Each fleet may contain up to six types of ships. These ships have unique characteristics that will determine the outcome of each engagement and ultimately the battle.

Ship Type	Health (HP)	Base Damage	Special Ability
Dreadnaught (DN)	750000	10000	0.1x Damage against bombers (B) and fighters (F).
Cruiser (CR)	500000	5000	0.2x Damage against bombers (B) and fighters (F).
Destroyer (DE)	250000	2500	--
Patrol Vessel (PV)	75000	1000	Instant Kill against fighters (F) or bombers (B).
Fighter (F)	2500	200	Instant Kill against Bombers.
Bomber (B)	2500	200	500x Damage against Cruisers (CR), and Destroyers (DE). 1000x damage against Dreadnaught (DN).

Your program will read from standard in and write results to standard out. There will be multiple battles described by the input data. Each battle will begin with a line containing the text “BEGIN BATTLE” the end of data for a battle will be denoted with a line containing the text “END BATTLE”. Within the battle there will be a line containing the text “ENEMY” the next line will contain a list of enemy ships. Your ships will be preceded by a line containing the text “FRIENDLY”. The formation of enemy ships is indicated by the order in which they are received. The list of ships will be comma delimited. The ship types will be indicated by the abbreviations (DN, CR, DE, PV, F, B). Below is an example of the input and output data.

Sample Input

```
BEGIN BATTLE
ENEMY
DN,CR,DN,PV,F,F,F,B,B
FRIENDLY
CR,CR,DE,DN,DN,PV
END BATTLE
BEGIN BATTLE
ENEMY
CR,CR,DE,DN,DN,PV
FRIENDLY
CR,CR,CR,PV,DE
END BATTLE
```

Sample Output

```
VICTORY IS ASSURED
RETREAT
```

7. The Beginning of the End

As a new employee* of Perceptive Cybernetics**, your first assignment is to construct a program that is capable of self-replication and modification. This is an important advancement for our artificial intelligence systems, as it would remove the cost of hiring hundreds of developers and testers to maintain the system. Our ethicists tell us that this is all perfectly fine and that nothing horrible can possibly come of this work. Besides, the lawyers say we are indemnified in case of any malfunctions of the software***.

Input: None

Output: The source text of the program that you submit as the answer with the following additional requirements

1) The source text of the problem must have one (and only one) line that contains the following line:

```
int x = 42;
```

2) The output must replace the numeric literal 42 with the numeric literal 666 on that line:

```
int x = 666;
```

Important tips

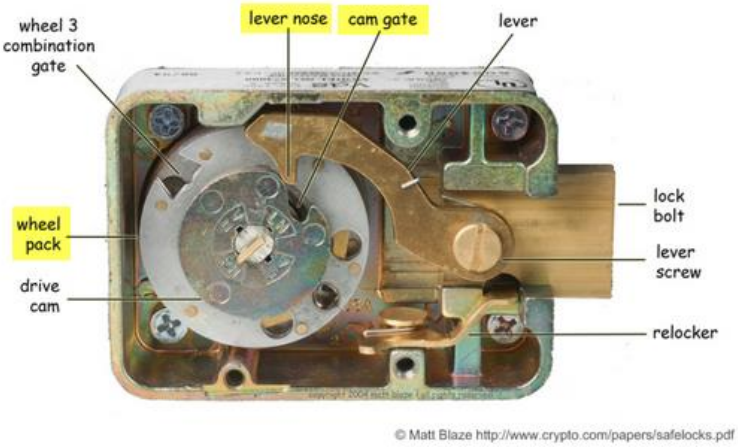
- You should use Unix-style line endings in the source code you submit and the output that your program generates.
- Your last line of source code should not contain a line ending.

** This is just a simulation... you have not actually been hired.*

*** Perceptive Cybernetics is not a real company, but that would be awesome.*

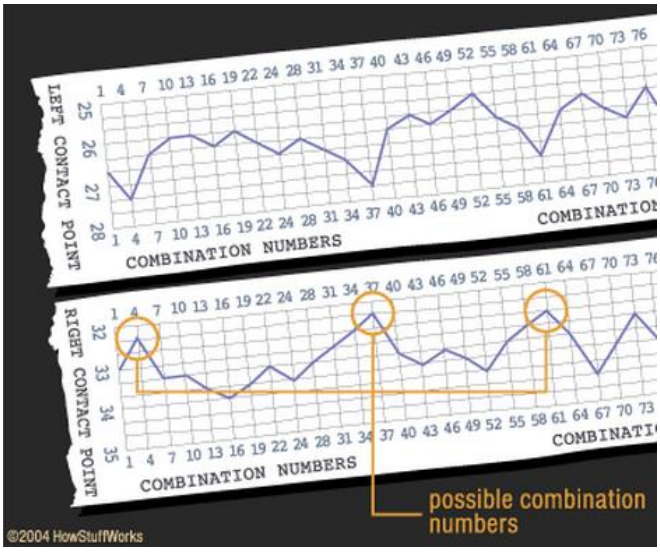
**** No lawyers were involved in the drafting of this problem. Thankfully.*

8. The Italian Job



In this popular movie, a technique called *lock manipulation* is used to perform safecracking of traditional (dialed combination) locks by hand. These locks will have a combination consisting of **w** numbers, where w is the number of wheels in the wheel pack. For example, when w=3, the combination might be 6-34-10. The number of *notches* on the dial is represented by **n**. This means that any number in the combination can be in the range of 0-**n**.

Using sense of feel, a safecracker will produce two graphs representing the left and right contact points for all notches* on the dial. A *contact point* is where the *lever nose* barely touches the beginning/ending of the *cam gate*. The points where the graphs are closest represent probable numbers in the combination. Note that while these numbers (or ones close to it) are likely to be in the combination, their order is unknown.



* Note: In reality, contact points are recorded to a sufficient accuracy of **every 3 notches**. This is because most locks have a *dialing tolerance* of up to ±1.25. Even if the tolerance is tighter, it is usually adequate to test only at every 3 notches, because the lever nose will be close enough to partially lower into the cam gate, release the lock bolt and open the safe.

Your task is to write a program that will help a safecracker analyze the information he collects. It will take the following input:

w - Number of numbers in the combination, which will be between 1 and 4.
n - Number of notches on the dial, which will be between 10 and 100.
LCP, RCP - Left and right contact points. These are represented as two sets of values which correspond to the possible combination numbers (starting at 1 and incrementing by 3).

The output should produce the points of convergence (where the pairs of left and right contact points are closest) along with a list of **w!** probable combinations, in order from lowest to highest. You can assume that there will be as many *unique* points of convergence as the number of wheels. You can also assume that each number in the combination is unique.

Sample Input

```
w:3
n:40
LCP:27.25,26.50,26.00,26.00,26.25,25.75,27.50,26.25,27.25,26.75,26.75,
26.50,27.00,26.75
RCP:32.25,33.00,33.25,33.50,34.00,33.00,33.25,33.25,33.50,33.25,34.00,
35.00,34.25,34.00
```

Sample Output

```
Your combination has 3 numbers from 0-40
Points of convergence are: 1,19,25
1-19-25
1-25-19
19-1-25
19-25-1
25-1-19
25-19-1
```

Sample Input

```
w:2
n:30
LCP:23.50,23.00,23.50,23.50,23.50,23.25,23.50,23.25,23.25,23.25,23.50
RCP:8.50,8.75,8.75,8.50,8.25,8.50,8.75,8.75,8.50,8.25,8.00
```

Sample Output

```
Your combination has 2 numbers from 0-30
Points of convergence are: 4,22
4-22
22-4
```