Problem A: Automated flooding

You are playing a computer game in which your goal is to flood the whole world.

The world is an $r \times c$ rectangle divided into unit square cells. Currently, all cells are dry. The game has two separate phases:

- In the first phase, you can use your mouse to flood some individual cells. Whenever you click a cell, it becomes a water cell. When you're done with clicking, you can start the second phase.
- In the second phase you just wait for the water to spread. Water spreads in steps. In each step, each dry cell that shares a side with at least two different water cells changes into a water cell.

Problem specification

You are given the dimensions of the world. Flood the entire world using **as few clicks as possible**.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of a single line with two positive integers r and c: the dimensions of the world. The individual cells of the world have coordinates ranging from $(0, 0)$ to $(r - 1, c - 1)$.

You may assume that $t = 100$ and that $1 \leq r, c \leq 100$.

In the **easy subproblem A1** you may also assume that $r = c$ (i.e., the world is a square).

Output specification

For each test case output a block of lines. The first line of the block should contain the number x of clicks you should make. Each of the following x lines should contain the coordinates of a cell you should click. There are usually multiple optimal solutions. We will accept any of them.

What to submit

Do not submit any programs. Your task is to produce and submit the correct **output files** `a1.out` and `a2.out` for the provided input files `a1.in` and `a2.in`.

Example

input

```
2

2 2

3 4
```

output

```
2
0 1
1 0
4
0 0
0 2
2 2
2 3
```

The second sample output is illustrated below, with 'W' denoting a water cell and '.' denoting a dry cell. On the left is the state of the world just after you made the four clicks. Progressing towards the right you see how the water spreads until it floods the entire world.

```
W.W.        WWW.        WWW.        WWWW        WWWW
....   ->   ..W.   ->   .WWW   ->   WWWW   ->   WWWW
..WW        ..WW        ..WW        .WWW        WWWW
```


Problem B: Bishopian paths

There is a rumor that William R. Hamilton played chess in his youth. Already in this young age he was obsessed by paths (and cycles) which visit each location exactly once.

One afternoon when he was staring at his chessboard he got intrigued by the bishop. He started to wonder whether it is possible to move the bishop to all chessboard squares of a given color so that the bishop visits each square exactly once.

Problem specification

A generalized chessboard is a rectangular grid of r rows by c columns. The color of grid squares alternates between black and white, like on a regular chessboard. In this problem the squares have coordinates ranging from $(1, 1)$ to (r, c), growing from top to bottom and from left to right. The square $(1, 1)$ in the **top left corner** is **always white**.

A bishop is a chess figure which is allowed to move along any diagonal by an arbitrary positive number of squares. Formally, a bishop located at (x, y) can move to any of the squares $(x+k, y+k)$, $(x+k, y-k)$, $(x-k, y+k)$ or $(x-k, y-k)$ for any $k > 0$. Note that from these rules it follows that the bishop always stays on squares of the same color.

A bishop's path is a sequence of chessboard squares such that each square (other than the first one) can be reached from the previous square by a bishop in one move. If a bishop's path visits each square of a given color exactly once, we call it a *bishop's Hamiltonian path*, or *bishopian path* for short. The path may start and end in any square of the given color.

You are given the chessboard dimensions and a color.

In the **easy subproblem B1** find a bishopian path or report that no bishopian path exists.

In the **hard subproblem B2** find a *non-intersecting* bishopian path or report that no *non-intersecting* bishopian path exists.

A non-intersecting bishopian path is defined as follows: If we draw the bishopian path as a polyline that connects the centers of visited squares, in order, the polyline must never overlap, cross, or even touch itself. (Formally, two consecutive line segments of the polyline can only share a single point, and any other two line segments must be completely disjoint.)

Input specification

The first line of the input file contains an integer $t \leq 600$ specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of a single line of the form "r c f", where r and c are dimensions of the chessboard and f is a character that is either `W` or `B`: the color of squares the bishop should visit. All test cases have $r, c \leq 125$ and $r \cdot c \geq 2$ (the chessboard has at least 2 squares).

Output specification

If there is no path of the desired type, output a single line with the text `impossible`.

Otherwise, for a path consisting of p squares output p lines each consisting of two integers r_i and c_i ($1 \leq r_i \leq r$ and $1 \leq c_i \leq c$): the coordinates of visited squares, in order.

In the example output below there is an empty line between the two test cases. But as with most IPSC problems, extra whitespace doesn't matter.

Example

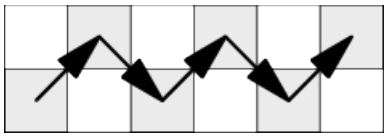
input

```
2

2 6 B

1 3 W
```

output

```
2 1
1 2
2 3
1 4
2 5
1 6

impossible
```

In the first example you are asked to find a path that visits all black squares of a chessboard with 2 rows and 6 columns. One possible solution (valid both for B1 and B2) is shown above. This output corresponds to the following diagram:



In the second example you are asked to find a path that visits all white squares on a chessboard with 1 row and 3 columns. There are two white squares and the bishop is unable to move between them, so no such path exists. Again, this answer is valid both for B1 and B2.

An example of a test case on which the answers for B1 and B2 differ is the test case "4 4 W".

Problem C: Collector's dilemma

Your local grocery store has started an innovative promotion: for every purchase, they give you a random emoji sticker. The person who collects the most distinct types of emojis wins free shopping for a year (excluding alcohol and gift cards). To keep the competition interesting, they decided not to disclose the total number of distinct emojis.

Your friend already has a sizable collection thanks to their love for pizza. To assess their chances, you would like to help them estimate how many different emojis are there to collect.

Problem specification

The store uses d types of emoji. The type of each emoji given to a buyer is selected uniformly at random from the set of all emoji types – i.e., each type is selected with probability $1/d$. All these random choices are mutually independent. The store has an unlimited supply of emojis of each type.

You know that before the promotion the store selected the value d at random, as described below.

You are given the multiset of emojis in your friend's collection. You are also given a closed interval $[a, b]$. Compute and return the probability that the actual value of d lies in the given interval.

The distribution of d differs in the easy and hard subproblem.

- In the **easy subproblem C1** you are given m: the maximum possible d. The actual value of d was selected uniformly at random from the set $\{1, 2, \ldots, m\}$.
- In the **hard subproblem C2** the value of d is not bounded from above, but smaller values of d are more likely. If x is a positive integer and p is a prime number, then the probability that $d = x$ is p^2 times more likely than the probability that $d = x \cdot p$. The minimum value of d is 1.

Input specification

The first line of the input file contains an integer $t \leq 100$ specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of two lines. The first line contains four space-separated integers n, a, b, m with the following meaning:

- n is the number of emoji types in your friend's current collection.
- a and b are the lower and upper bound of the query interval.
- In **C1**, m is the maximum possible number of distinct emojis. In **C2**, m is zero.

The second line of each test case consists of n integers $x_1, \ldots, x_n$: the number of collected emojis of each type.

Note that the emojis don't have any serial numbers, so you should **not** interpret x_1 as "the number of emojis of type 1". The order of the x_i does not matter. For example, if the second line of a test case is "1 1 2 1", the whole information it contains is that your friend currently has 5 emojis: a pair of identical ones and three unique ones.

In the **easy subproblem C1** we have $1 \leq n \leq 20$, $n \leq m$, and $1 \leq a \leq b \leq m \leq 10^4$. All x_i are positive and their sum does not exceed 40.

In the **hard subproblem C2** we have $1 \leq n \leq 40$, $1 \leq a \leq b \leq 10^{18}$, and $b - a \leq 20\,000$. The value of m is always 0, indicating that there is no upper bound on the number of emoji types. All x_i are positive and their sum does not exceed 120.

Output specification

For each test case, output a single line with a floating-point number p: the probability of d being in the closed interval $[a, b]$, given the known distribution of d and your observations.

Make sure that for non-zero p your output contains at least seven (preferably more) **most significant decimal digits** of p. Your p will be considered correct if it is within the interval $[0, 1]$ and its **relative** error is at most 10^{-6}.

The value of p may be printed in scientific notation ("`1.2345678E-90`").

Note that (as opposed to many other problems that involve floating-point numbers) we are **not** looking at the **absolute** error of your answer.

Example

input

```
5

1 1 1 2
2

10 10 100 10000
1 1 1 1 1 1 1 1 1 1

5 1 4 10000
4 6 5 2 1

15 150 5000 10000
1 1 2 2 1 1 1 2 2 1 3 1 1 1 1

5 10 200 0
1 2 2 1 3
```

output

```
0.6666666666667
0.0034745158491
0.0000000000000
0.0027270798056
0.1182410425858
```

The first four sample test cases (with positive values of m) can only appear in the easy subproblem. The last sample test case (with $m = 0$) can only appear in the hard subproblem.

In the first sample test case you know that the store flipped a fair coin to decide whether the number of emoji types is 1 or 2. Your friend has two emojis and both happen to be of the same type. While it is still possible that there is a second type, it is more likely that this collectors' game is a bit boring – every emoji is the same.

In the second example you know that the actual number of emoji types was chosen from the range 1-10000. You are asked what is the probability that there are between 10 and 100 kinds of emojis, given that your friend has collected ten emojis so far and every one of them turned out to be distinct. Receiving ten distinct emojis is not so likely if the total number of emoji types is small, so you conclude that the probability of d being in the given interval is quite low.

In the third case you have already observed five different types of emoji. Thus, you can be absolutely certain that d does not lie in the interval [1,4].

Problem D: Dazzling digits

Numbers with repeated digits are boring. Look at them: 44 is boring, 474 is boring, 1 000 000 is sooooo boring it hurts.

Numbers without repeated digits aren't boring at all. As you read such a number, you will always encounter something new! Look: 52 107 is pretty, and 9 087 432 156 is among the most pleasant numbers one can read.

Problem specification

You are given a positive integer n.

Write n in the form $a_1 + \cdots + a_k$. Each of the a_i must be a positive integer. The values a_i must not be boring. Their count (i.e., the number k) must be as small as possible.

The values a_i don't have to be distinct. The same digit may appear in multiple a_i.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of a single line with a single positive integer n.

In the **easy subproblem D1** we have $t = 1000$ and $1 \leq n \leq 10\,000$.

In the **hard subproblem D2** we have $t = 30\,000$ and $1 \leq n \leq 2\,000\,000\,000$.

Output specification

For each test case, output a single line of the form "$k\ a_1\ \ldots\ a_k$".

If there are multiple optimal solutions, you may output any of them.

Example

input

```
3

123

99

999123456
```

output

```
1 123
2 90 9
2 20469135 978654321
```


Problem E: Elevation alteration

You're a rich tycoon who owns an enormous transportation company. Your current project is a new railroad that will go across the whole country and bring you grossly huge gross profit. But before you can start building the railroad itself, there are terrain adjustments to be made.

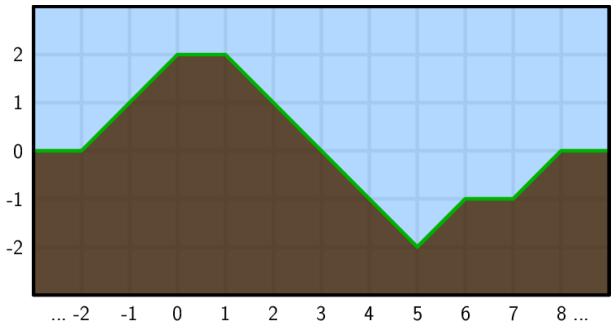
On a satellite image (i.e., looking from above), the future location of the railway appears as an infinite straight line. But you're more interested in the terrain's elevation profile (i.e., looking at the side view). Right now, everything is completely flat. That's boring! You need some more interesting scenery so that tourists will pay you lots of money. Since you're so rich, you decided to make some hills and valleys.

Problem specification

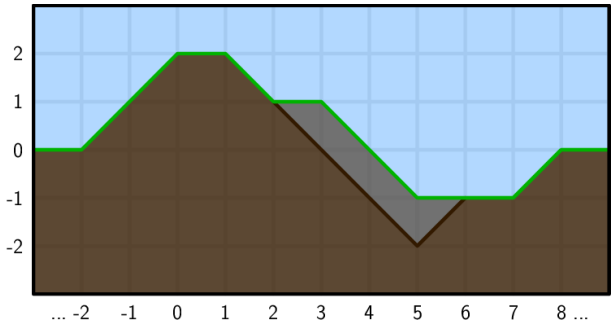
You took the satellite image and made a mark every 1 meter. You assigned those marked points coordinates from negative infinity to positive infinity.

The elevation (terrain height) at every point is an integer amount of meters. Between them, the ground is always a straight slope connecting the nearest two points. Your engineering department has informed you that train engines can't handle very steep slopes, so the difference in elevation between two neighboring points can be at most 1 meter.

For example, part of an elevation profile that is suitable for trains could look like this:



Each change to the terrain must also leave it suitable for trains. This means that whenever raising a point would cause it to be too far away from its neighbor, you must also raise that neighbor. Lowering a point works the same. For example, if you want to raise point 3 by one meter, you must also raise points 4 and 5 as shown below:



The cost of raising or lowering terrain is equal to the number of square meters of dirt you added or removed. In our example, to raise point 3 you had to add 3 square meters of dirt.

Initially, all points have the same elevation. You have given your construction workers a list of commands to increase or decrease the elevation of various points by 1 meter. The workers will complete

the commands in the given order, one at a time (each time also raising or lowering other points if necessary). Now you'd like to know the cost of each command.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

The first line of each test case contains the number of commands n. The i-th of the following n lines contains two integers p_i and e_i describing the i-th command: the coordinate of the point to be changed, and whether to raise it by 1 meter ($e_i = 1$) or lower it by 1 meter ($e_i = -1$).

In the **easy subproblem E1**, $1 \leq n \leq 1\,000$ and $-1\,000 \leq p_i \leq 1\,000$.

In the **hard subproblem E2**, $1 \leq n \leq 5\,000\,000$ and $-10^9 \leq p_i \leq 10^9$.

Because `e2.in` is about 60 MB, you cannot download it directly. Instead, we have provided a small Python 2 program `e2gen.py` that will generate `e2.in` when executed. The generator should take under 1 minute to run on average hardware. We recommend running it early – for example, starting it as you start working on your solution for this problem.

Output specification

For each test case: Let c_i be the cost of command i, for all $1 \leq i \leq n$. (Note that all c_i are positive integers.) Then, let $d_i = i \cdot c_i$. Output one line with a single number: $(d_1 + \cdots + d_n) \bmod (10^9 + 9)$.

Example

input

```
1

8
0 1
5 -1
1 1
5 -1
1 1
0 1
7 -1
3 1
```

output

```
71
```

The costs are 1, 1, 1, 3, 2, 2, 1, 3. The last command corresponds to the change from the first to the second picture in the problem statement.

Problem F: Flipping and cutting

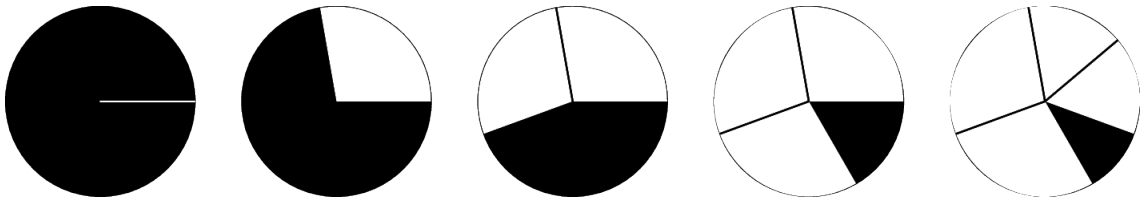
On the table we have a perfect circle cut out of paper. The circle is all black from the top and all white from the bottom. The circumference of the circle is a positive integer c.

We also have a pair of scissors. We will use the scissors to cut the circle. At the beginning we will choose an arbitrary point on the boundary of the circle (we'll call it the *active point*) and we'll make a straight cut from that point to the center of the circle.

Next, we will choose a positive integer s (with $s < c^2$): the square of our step length. Finally, we are going to perform an infinite sequence of rounds. Each round consists of a few simple steps:

1. Find a *new point* by starting at the current active point and measuring the distance $\sqrt{s}$ along the boundary of the circle, going counter-clockwise.
2. Make a straight cut from the new point to the center of the circle.
3. We now have a wedge (a sector of the circle) that starts with the cut from the active point to the center and continues counter-clockwise all the way to the cut from the new point to the center. Note that the inside of the wedge may also contain other cuts.
4. We carefully lift the entire wedge from the table, flip it upside down and return it back to its place. Note that if the wedge already consisted of multiple pieces, their order is reversed by this operation.
5. The new point becomes the active point for the next round.

The figure below illustrates the first four rounds for $c = 360$ and $\sqrt{s} = 100$. Pay close attention to what happens in the fourth round.



Problem specification

For some pairs (c, s) it may happen that after finitely many rounds we will have a completely black circle again.

Find out which pairs (c, s) have this property, and for each such pair determine the smallest number of rounds after which it happens.

(A technical note: A circle is completely black if each point of the top side of the circle is black or lies on one of the cuts. In other words, you don't have to worry about the color of the points that lie directly on the cuts.)

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of a single line containing the positive integers c and s. As stated above, you may assume that $s < c^2$.

In the **easy subproblem F1** you may also assume that $c \leq 1\,000$ and that $\sqrt{s}$ is an integer.

In the **hard subproblem F2** you may assume that $c \leq 10^6$.

Output specification

If the process never returns to a completely black circle, output a single line with the text "`never`". Otherwise, output a single line with a single integer: the smallest positive number of rounds after which the circle will again be completely black.

Example

input	output
3 360 3600 360 10000 360 90000	12 24 4

In the first example case we have a circle with circumference 360 units and we are cutting wedges that each contain 60 units of the circle's circumference. Clearly, after six rounds we will have flipped each part of the circle exactly once, and after another six rounds everything will be back to where we started.

The second example input corresponds to the situation shown in the figure above. It takes 24 rounds to get us back to an all-black circle.

In the third example we flip 5/6 of the circle in each round. Maybe surprisingly, already after four rounds the circle will be completely black again.

Problem G: Gunfight

It's almost the end of the film! All the heroes and villains are in a Mexican standoff, pointing guns at each other and tensely waiting for the final shoot-out. Huge secrets are being revealed! Alliances are shifting! Who will live and who will die? Nothing is certain.

Problem specification

Every character has a gun and can aim it at one other character. All the characters have perfect aim, so when they fire, the person they're aiming at certainly dies. Everyone also has superhuman reaction time, so in the instant before they die, they also fire their gun. The next person also squeezes the trigger and then dies, and so on, until it reaches someone who wasn't aiming at anyone or someone who's already dead.

Initially, nobody is aiming at anyone. During the standoff, two things can happen:

- Character a hears a shocking secret and aims their gun at character b (or at nobody).
- You become curious: if character a fired right now, would character b die?

Input specification

The input file for each subproblem consists of one single test case.

The first line of the input file contains an integer n specifying the number of characters and an integer q specifying the number of events. The characters in the film are numbered from 1 to n.

Each of the next q lines contains either "1 a b" ($1 \leq a \leq n, 0 \leq b \leq n$), which means that character a pointed their gun at character b (or at nobody if b is zero), or "2 a b" ($1 \leq a, b \leq n$), which means that you want to know whether b would die if a would fire.

In the **easy subproblem G1** you may assume that $n = 3\,000\,000$ and $1 \leq q \leq 10\,000\,000$, and also that there will never be a group of characters aiming at each other in a cycle.

In the **hard subproblem G2** you may assume that $n = 5\,000\,000$ and $1 \leq q \leq 40\,000\,000$.

Do not assume anything else about the events. In particular, it's possible that character a was already aiming at character b when you process an event "1 a b". In that case nothing changes. And in the hard subproblem, if a revealed secret is particularly shocking, a character could even point the gun at themselves.

Because `g1.in` and `g2.in` are about 700 MB in total, you cannot download them directly. Instead, we have provided small Python 2 programs `g1gen.py` and `g2gen.py` that will generate `g1.in` and `g2.in` when executed. Each generator should take under 8 minutes to run on average hardware. We recommend running them early – for example, starting them as you start working on your solution for this problem.

Output specification

For each question, answer 0 if the character would survive and 1 if the character would die. Concatenate the answers into one long binary number, so that the last (least significant) digit is the answer to the last question. Convert this number to decimal and print it modulo $10^9 + 9$.

Example

input

```
3 12
2 1 2
1 1 2
1 2 3
2 1 3
2 3 1
2 2 2
1 3 1
2 3 3
1 1 0
2 3 3
1 3 3
2 3 3
```

output

```
37
```


Problem H: Holy cow, Vim!

Little Johnny is attending a summer programming camp. The very first assignment was to write a program that reads a number x and prints the same number x. Johnny's program was already working, but then an accident happened. While using the Vim editor, Johnny pressed something without paying attention and his program got turned upside down. That is, he somehow reversed the order of lines in his program.

To Johnny's amazement, the program still worked, but now it did something different: it read x and printed x^2.

Johnny tried to remember what he pressed to put the lines back in the correct order, but he made another mistake and Vim sorted the lines of his program. Johnny tried the new program and was completely lost for words: the program now read x and printed $-x$.

"Holy cow, Vim is magic! I'll use it until the end of my life!" exclaimed Johnny.

"That's just because nobody knows how to exit it," another student yelled back.

Can you write a program that behaves the same way as Johnny's program?

Problem specification

In this problem, you'll use a simple stack-based programming language. The memory is a stack of signed integers. Various commands push values onto the stack or pop values from the top of the stack. The stack is initially empty and may be left non-empty when the program ends.

A program consists of several *lines*, and each line consists of one or more *commands* separated by semicolons. A command can be one of the following:

- "`input`": Reads the number x from the input and pushes it onto the stack. You may only execute "`input`" once per an execution of your program.
- "`jump` j": Immediately jumps to the beginning of line j. Lines are numbered counting from 0 to $n-1$ where n is the number of lines. Jumping to $j = n$ exits the program. Jumping to $j < 0$ or $j > n$ is an error.
- "`pop`": Removes the top element from the stack. Results in an error if the stack is empty.
- "`print`": Removes the top element from the stack and prints its value. Results in an error if the stack is empty. You may only execute "`print`" once per an execution of your program.
- "`push` p": Pushes the constant p onto the top of the stack.
- "`dup`": Duplicates the top element of the stack. If the current top element is t, "`dup`" does the same thing as "`push` t". Results in an error if the stack is empty.
- "`+`", "`-`", "`*`" and "`/`": Pops the top element a from the stack, then pops the next element b from the stack, then pushes $a + b$, $a - b$, $a \cdot b$, or a/b rounded towards zero, respectively. Results in an error if the stack contains fewer than two numbers. Division by zero is also an error.

The language is very strict. You cannot use any extra whitespace or semicolons or anything like that.

Only integers between -2^{31} and $2^{31} - 1$ (inclusive) are supported. Pushing an integer outside of this range to the stack is an error.

Input specification

There is no input.

Output specification

Your task is to write a program that reads an integer x and outputs x. However, if the order of the lines of the program is reversed (i.e., the last line becomes the first line, etc.), the new program should output x^2. And if the lines of the program are sorted lexicographically, it should output $-x$.

You may assume that $|x| \leq 30\,000$.

The program can have at most 1000 lines. For any valid x your program must terminate after the execution of at most 10 000 commands.

In the **easy subproblem H1** each line may contain **up to 1000 commands**.

In the **hard subproblem H2** each line may contain **at most two commands**.

Example

The following program reads x and outputs $x - 7$.

```
push 7
input;-
print
```

If you reverse the order of lines of the above program, "print" will become the first command, and the program will fail because it tries to print the top of an empty stack.

Problem I: Internet problem

Lisa and Sarah have exposed a massive conspiracy and now they're on the run from the corrupt government. Being together makes it too risky that they could both be captured, so they have to communicate through the Internet. But the normal Internet isn't safe enough, so they send each other secret messages through the dark web.

On the dark web, every message can take a long and convoluted path through many servers until it reaches its destination, and it might even go through one server multiple times. This makes messages much harder to trace.

But Lisa is still worried. What if the government has already hacked one of the servers of the dark web? If the hacked server is in a good central location, it could intercept all of her messages to Sarah, regardless of what path they take.

Help Lisa solve her Internet problem!

Problem specification

The dark web consists of n servers, numbered from 1 to n. The servers are connected by m network links. Links are directed – if one server can transmit a message to another, the opposite doesn't have to be true. Lisa is connected to server 1 and Sarah is connected to server n. Whenever Lisa wants to send a message to Sarah, she chooses a route for the message: a sequence of consecutive network links that goes from server 1 to server n. The route may go through each server multiple times.

The government wants to intercept Lisa's messages to Sarah. They can hack one server so it records all messages that go through it. They want to see every message from Lisa to Sarah **exactly once**. ("At least once" is needed so they learn all about their plans, and "at most once" is needed so their hard drives don't fill up with duplicates.)

Find all the servers which satisfy that condition.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

The first line of each test case contains the integers n and m. Each of the next m lines contains two integers a, b ($1 \leq a, b \leq n$) meaning that server a can transmit messages directly to server b. (It may be the case that $a = b$.) Each distinct ordered pair a, b will be given at most once.

In the **easy subproblem I1**, $2 \leq n \leq 1\,000$ and $0 \leq m \leq 3\,000$.

In the **hard subproblem I2**, $2 \leq n \leq 500\,000$ and $0 \leq m \leq 1\,000\,000$.

Output specification

For each test case, output two lines. On the first line, print the number of servers that could be hacked by the government. On the second line, print a space-separated list of numbers of those servers, in the order in which messages from Lisa to Sarah go through them.

Note that for some test cases there may be no path from Lisa to Sarah. If that is the case, output an empty set of servers.

Example

input

```
4

4 3
2 4
1 3
3 2

2 2
1 2
2 1

3 1
2 3

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

output

```
4
1 3 2 4
0

0

2
1 4
```

First test case: All messages must take the same path, so the government could hack any server on the path.

Second test case: The government doesn't want to get any duplicates, so they can't hack either server. Lisa and Sarah are safe.

Third test case: Lisa can't send any messages anyway, so there is nothing to hack.

Fourth test case: The government can't hack both 2 and 3 at once. If they hack only 2, or only 3, they won't see all messages.

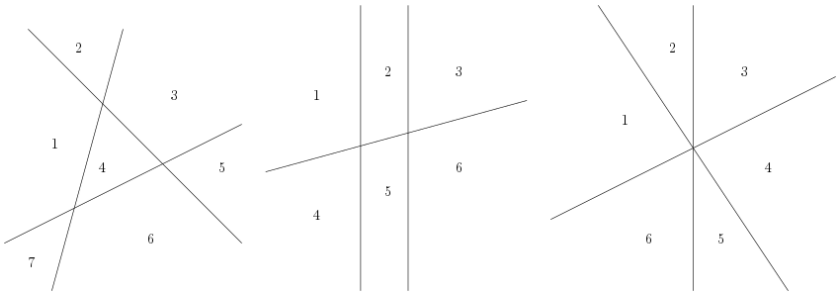
Problem J: Judicious cuts

"Draw seven red lines, all of them strictly perpendicular to each other, some of them with green ink and some of them with transparent ink. Also, one of them should be in the form of a kitten."
(The Expert)

Oh, don't worry, your task is much simpler. We'll stay in the two-dimensional plane.

A straight line cuts the two-dimensional plane into two regions. With two such lines you can cut the plane either into three regions (if you use two parallel lines) or into four regions (if you use lines that intersect each other).

Below are some ways of arranging three lines to cut the plane into 7, 6, and 6 regions, respectively.



Problem specification

We want to divide the plane into r regions. Tell us how to draw the lines.

In the **easy subproblem J1** you may output any set of lines that divides the plane into exactly r regions and satisfies the output format described below.

In the **hard subproblem J2** you must also divide the plane into exactly r regions, but this time you must use as few lines as possible.

Input specification

The first line of the input file contains an integer $t \leq 1000$ specifying the number of test cases. Each test case is preceded by a blank line.

Each test case is a single line with a single integer $1 \leq n \leq 1000$: the desired amount of regions.

Output specification

The output for each test case should start with a line with a single integer ℓ: the number of lines you want to draw. Then, output ℓ lines, each describing one line in the plane. Each line is specified by two space-separated integers m and b. These represent the line with the equation $y = mx + b$. (You are not allowed to draw vertical lines. Obviously, you don't need to do so.)

For every test case you can use at most 1 000 lines. All slopes (m) and y-intercepts (b) have to be between $-10\,000$ and $10\,000$, inclusive. It is guaranteed that such solutions exist for all valid test cases.

Example

input

```
3

2

4

35
```

output

```
1
0 5
2
2 0
-2 4
10
2 0
1 1
0 1
0 2
0 3
0 4
0 5
0 6
-1 3
-2 4
```

In the first sample, there is just one line, and that always divides plane into 2 regions. In the second sample, two intersecting lines divide plane into four quadrants. The solution shown for the third case is not optimal, and hence it would not be accepted in the hard subproblem.

Problem K: Kirk or Picard?

"Only question I ever thought was hard was do I like Kirk, or do I like Picard?"
—Weird Al Yankovic: White and Nerdy

If you're in a hurry and you don't have a good pseudorandom generator, you can try asking a Trekkie whether they prefer Kirk or Picard. Or ask them whether they prefer to be called a Trekker instead. And in case there is no Trekkie around, simply breathe in through your nose and check which nostril carried most of the air.

This task is also about imperfect pseudorandom generators. We'll give you most of the output and your task is to fill in the blanks – literally.

Problem specification

We have generated two sequences of pseudorandom bits. Each sequence has been generated by using a specific linear congruential pseudorandom generator (not necessarily a good one). For more information, see the Wikipedia page about linear congruential generators.

You are given those two sequences. Each consists of 4 000 000 values from the set $\{0, 1, 2\}$. 0 and 1 are actual bits output by a pseudorandom generator. 2 is a blank – it represents a bit with an unknown value. Each sequence contains exactly 4 000 blanks.

In the **easy subproblem K1** choose either of the two sequences and guess at least 2 500 of the blanks correctly.

In the **hard subproblem K2** guess at least 2 500 blanks in each of the two sequences correctly.

Input specification

The input file `k.in` contains two lines, each describing one of the sequences. Each sequence is given in base-81 encoding. More precisely, the sequence is divided into groups of four values and a group (a, b, c, d) is encoded into the character with ASCII value $(33 + 27a + 9b + 3c + d)$.

Output specification

In the **easy subproblem K1** output a single string of exactly 4 000 zeroes and ones: your guess for the contents of the blanks in one of the sequences. The guesses correspond to the blanks in the order in which they appear in the given input sequence.

Your output will be accepted if it has enough correct guesses for either of the two input sequences. (You do not have to indicate which sequence is the one you chose.)

In the **hard subproblem K2** output two whitespace-separated strings, each consisting of exactly 4 000 zeroes and ones. The first string is your guess for the first sequence given in the input, and the second string is your guess for the second input sequence. Your output will be accepted if both guesses are good enough.

Example

input	output
`++2+a1`	`0110`

The sample input encodes the sequence 0101010101220101210101210121.

This sequence is not really random, it looks like it's just alternating zeroes and ones. Thus, the missing bits seem to be 0, 1, 0, and 0. If that was indeed the case, the sample output would be 75% correct. Getting 2500 out of 4000 correct means having 62.5% of your guesses correct.

Problem L: Lucky draws

Yvonne and Zara are playing a card game. The game is played with a standard 52-card deck. In this game we don't care about specific ranks or suits, only about color (red or black). So it is enough to note that the deck contains exactly 26 red and 26 black cards.

Before the game Yvonne and Zara agree on a positive integer k. The game is then played as follows:

1. Yvonne takes the deck, removes any k cards and lays them out into a sequence in front of her.
2. Zara takes the remainder of the deck, removes any k cards and lays them out into a sequence in front of her. Zara is not allowed to choose the same sequence of red and black cards as Yvonne.
3. The girls shuffle the remainder of the deck.
4. The girls deal cards from the top of the deck into a sequence. If at any moment the colors of the last k cards dealt from the deck match Yvonne's sequence, Yvonne wins. If the last k cards match Zara's sequence, Zara wins.
5. If they run out of cards in the deck and neither girl has won the game, they decide the winner by flipping a fair coin.

Example of gameplay

- Yvonne chooses the sequence "red, red, black".
- Zara chooses the sequence "black, black, black".
- The remainder of the deck (24 red and 22 black cards) is shuffled.
- The girls start dealing cards off the top: red, black, black, red, red, red, black.
- At this moment the last three cards are "red, red, black", which is precisely Yvonne's sequence. The game is now over – Yvonne won.

Problem specification

You are given the value k. Assume that each girl wants to maximize her probability of winning and that they both play the game optimally.

In the **easy subproblem L1** determine which girl is more likely to win the game.

In the **hard subproblem L2** determine the probability of Yvonne winning the game.

Input specification

There is no input.

Output specification

Output exactly 26 lines, corresponding to $k = 1, 2, \ldots, 26$.

In the **easy subproblem L1** each line should contain the name of the girl who is more likely to win the game (either "`Yvonne`" or "`Zara`"). In case they are both equally likely to win, print the string "`tie`" instead.

In the **hard subproblem L2** each line should contain the probability that Yvonne wins the game for that particular value k, assuming both girls play the game optimally. Output at least 10 decimal places. Solutions that differ from our answer by at most 10^{-9} will be accepted as correct.

Example

Both for $k = 1$ and for $k = 2$ the correct output for L1 is "`tie`" and the correct output for L2 is "`0.5000000000`".

Even though the game seems advantageous for Yvonne, because she gets to choose her sequence first, there isn't much she can do if the sequences are short. For $k = 1$ there is essentially only one possibility: Yvonne chooses one color, Zara chooses the other color, and the first card off the deck decides the game.

For $k = 2$ one optimal choice for Yvonne is the sequence "red, black", and then an optimal choice for Zara is the sequence "black, red". It should be obvious that the resulting game is symmetric, which means that each girl wins it with probability 0.5.

Problem M: Matching pairs

The input file contains a coordinate grid with 10×10 similar-looking shapes. Among those 100 shapes there are exactly three matching pairs of shapes:

- Two shapes are identical. (One can be obtained from the other by rotation and translation only.)
- Two shapes are mirror images of each other. (They are not identical as defined above, but one can be obtained from the other by flipping it once and then rotation and/or translation.)
- Two shapes are complements of each other. (Explained below.)

As you can see from the input file, the shapes are actually pictures of simple graphs on 7 vertices. "Complements" should be interpreted in that sense. Two shapes are complements of each other if one of them can be rotated and translated – without flipping – and placed on top of the other in such a way that their vertices coincide and each pair of vertices is connected by an edge in *exactly one* of the two graphs.

Problem specification

In order to solve the **easy subproblem M1**, find any one of the three matching pairs.
In order to solve the **hard subproblem M2**, find all three matching pairs.

Input specification

The input is a PNG picture of the shapes. The input is the same for both subproblems.

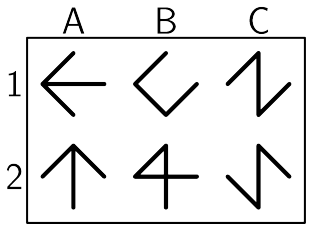
Output specification

For the **easy subproblem M1** output one line with two space-separated tokens: the coordinates of two shapes that form any one of the three matching pairs. Coordinates of a shape should be printed in the form "LN", where L is a letter and N is a number.

For the **hard subproblem M2** output three lines, each containing two space-separated tokens: the coordinates of the two shapes that form one of the three matching pairs. Output each matching pair exactly once.

The order of the two shapes within a matching pair does not matter. The order of lines in the output for M2 also does not matter.

Example



The example input shown above contains six graphs on four vertices. The three matching pairs are the following ones:

- The identical shapes are A1 and A2. You can rotate A1 by 90 degrees to the left to get A2.

- The mirror images are C1 and C2. They are not identical but C2 is a vertical mirror image of C1.
- The complementary shapes are B1 and B2. You can rotate B2 by 90 degrees to the right and place it on top of B1 to get a complete graph – a square with both diagonals. Note that each of the six edges of the complete graph is present in exactly one of the shapes B1 and B2.

One possible output for the easy subproblem:

```
C2 C1
```

One possible output for the hard subproblem:

```
A1 A2
B1 B2
C1 C2
```

Fun fact

The same problem but with fewer (49 instead of 100) different shapes was used in one of the rounds of the 2016 World Puzzle Championship. However, this is IPSC and bigger is always better, right? :)

