IPSC 2017

problems and sample solutions

Problem Q: Queue skipping

Recently there was a rumor that on Monday the meat store will actually have some meat. It's Monday, half past one in the morning. The entire town is already waiting for the store to open. More precisely, there are n people waiting in a line. The people are numbered 1 through n in line order, with person 1 being the one currently closest to the door of the store.

During the next few hours, at some moments one of the people will use some trick in order to skip to the beginning of the queue. Examples of such tricks include:

- Look over there, isn't that *[famous TV show host]*?
- Please let me through, I'm with a small child!
- I stood here before, I just went to the bathroom.
- Hey, look at that guy trying to skip the queue, I'll go and throw him out!

Problem specification

You are given the sequence of people skipping to the front of the queue. After all those events, the meat store will open. People will enter the store one at a time. Inside, they will learn that the meat didn't arrive (again!), so they will go home empty-handed.

Find out who will waste the most time waiting for nothing. In other words, find out who will be the last person in the queue after all the skipping to the front is over.

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 starts with a line containing two positive integers n and e: the number of people and the number of events. The rest of the test case consists of e lines, each containing a single positive integer: the number of a person who just moved to the front of the queue.

In the **easy subproblem Q1** you may assume that $t = 100$ and $1 \leq n, e \leq 100$.

In the **hard subproblem Q2** you may assume that $t = 100$ and $1 \leq n, e \leq 10^5$.

Note that you cannot download the input for subproblem Q2 directly. Instead, we have provided a small Python 2 program that will generate the file `q2.in` when executed.

Note

In the real contest, some large input files may be provided in the same way as the input `q2.in` in this practice problem. Please make sure you are able to generate it.

Output specification

For each test case output a single line with a single positive integer: the number of the last person in the queue when the meat store opened.

Example

input	output
2 3 4 3 1 3 2 7 1 1	1 7

In the first example test case, the queue changed as follows:

- *at the beginning: (1,2,3)*
- *person 3 skips to the front: (3,1,2)*
- *person 1 skips to the front: (1,3,2)*
- *person 3 skips to the front again: (3,1,2)*
- *person 2 skips to the front: (2,3,1)*
- *thus, the last person at the end is person 1.*

In the second example test case, person 1 skipping to the front of the queue doesn't change the queue at all. In particular, the last person in the queue is still person 7.

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'mišof' Forišek
Quality assurance:	Mário Lipovský

Solution

The easy subproblem is easily solved by brute force: just simulate the process described in the problem statement. And if you used a reasonably fast programming language and had a bit of patience, you could actually use the same code to solve the hard subproblem as well – the test cases aren't that large, a well-implemented solution that runs in $O(ne)$ time should solve the hard subproblem within a few minutes.

But where's the fun in that? We want a more clever solution!

In order to solve the hard subproblem in an efficient way, realize that there are only two possibilities for the answer:

1. If there are some people who never skipped to the front, these are now the people at the end of the queue. In particular, the last person in the queue is the one with the largest number among the people who never skipped the queue.
2. If everybody skipped to the front at least once, the person at the end of the queue is the one who did so least recently.

Once we make this observation, we can easily solve the problem in linear time – more precisely, in $O(n + e)$ steps.

A small bonus challenge: can you compute not just the last person but the entire final state of the queue in $O(n + e)$ time?

Problem R: Russian roulette

You went out to celebrate with your $n-1$ friends. You all got drunk and decided that playing Russian Roulette with a paintball gun sounds like a great idea.

Problem specification

The gun is a revolver with c chambers ($c \geq n$). Exactly $n-1$ of those chambers now contain a paintball, all others are empty.

The game has one other parameter: a positive integer constant k.

At the beginning of the game all n players stand around a circle. We will number the players 0 through $n-1$ as they're standing in clockwise order. Player number 0 takes the gun. The game is then played by repeating the following three steps until the gun becomes empty:

1. The person who has the gun uses it on themselves. This means that they spin the cylinder to choose one of the chambers uniformly at random and then they point the gun on themselves and pull the trigger.
2. If the chosen chamber was empty, there are exactly k rounds of passing the gun. In each round the person who has the gun passes it clockwise to the next person who is still playing the game.
3. If the chamber contained a paintball, the person is hit by the paintball and loses the game. In this case, they just pass the gun to the next playing person clockwise and they leave the game.

The winner is, obviously, the only person who doesn't get shot.

You are given the parameters n, c, and k. Compute where you should stand at the beginning of the game in order to maximize your probability of winning. Also, compute the exact value of that probability.

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 case consists of a single line with the integers n, c, and k. You may assume that $1 \leq k \leq n \leq c$.

In the **easy subproblem R1** you may assume that $c \leq 10$.

In the **hard subproblem R2** you may assume that $c \leq 1000$.

Output specification, easy subproblem R1

For each test case, print one line containing two space-separated numbers: your position at the beginning of the game which maximises the probability of winning and the probability of winning when standing at that position.

Output the probability as a floating-point number with at least five decimal places. Answers with absolute or relative error up to 10^{-4} will be accepted.

You may assume that in each test case the optimal position is unique and that the probability of winning for the optimal position differs from the probability for any other position by more than 10^{-6}.

Output specification, hard subproblem R2

For each test case, print one line containing two space-separated numbers: your position at the beginning of the game which maximises the probability of winning and **a number that encodes** the probability of winning when standing at that position, as defined below.

If there are multiple optimal positions, output the **smallest** one of them.

Let the maximum probability of winning be an irreducible fraction of the form p/q. In order to output this probability, print the number pq^{-1} modulo $(10^9 + 9)$, where q^{-1} is the modular inverse of q modulo $10^9 + 9$. It is guaranteed that the maximum probability of winning can be expressed as a fraction and that the modular inverse of q exists.

Example, easy subproblem R1

input

```
3

3 3 1

2 2 2

4 5 3
```

output

```
2 0.5076923077
1 1.0000000000
2 0.4105090312
```

In the first example, each player passes the gun to the next person clockwise. Intuitively, you should go as late as possible – stand counter-clockwise from the person who starts. This turns out to be the optimal strategy. The exact probability of winning for player 2 *is* 33/65.

In the second example, player number 0 *keeps on using the gun until they lose the game. (After each unsuccessful shot the gun gets passed twice, which brings it back to player* 0.*) Hence, the probability of winning is* 1 *for player* 1 *and* 0 *for player* 0.

In the third example, the following happens, in order:

- *While there are four players, each time they pass the gun clockwise three times, it is as if they passed it counter-clockwise once. The gun essentially travels counter-clockwise around the circle until a person shoots themselves.*
- *The next person clockwise gets the gun. This person now keeps trying to shoot themselves until they succeed – which they eventually will.*
- *The remaining two people take alternating shots until one of them loses.*

The probabilities of winning for players 0, 1, 2, 3 *are* 40/609, 100/609, 250/609, *and* 219/609, *respectively.*

Example, hard subproblem R2

For the sample input shown above the correct output for subproblem R2 looks as follows:

```
2 661538468
1 1
2 348111662
```

In the first test case the exact answer is 33/65. *Since* $65^{-1} \equiv 323\,076\,926 \bmod (10^9 + 9)$, *the second number in the first line of the sample output is* $(33 \cdot 323\,076\,926) \bmod (10^9 + 9) = 661\,538\,468$.

Task authors

Problemsetter:	Monika Steinová
Task preparation:	Jakub 'Xellos' Šafin
Quality assurance:	Edo 'Baklažán' Batmendijn

Solution

In the easy subproblem the required precision was quite low, so you should be able to solve it simply by simulating a lot of games. We will now show how to solve the hard subproblem.

Note that in order to describe the state of the game you only need to know the number of people remaining, your position among them, and the person currently holding the gun. In fact, it is sufficient to assume you're always at position 0 with i other people in the circle and the gun at position j. Thus, there are only $O(n^2)$ states.

The obvious choice now is to use dynamic programming to compute the probability of your victory for each state: `prob[i][j] = p_suc[i][j] + (1-p) * prob[i][(j+k)%(i+1)]`. The term `p_suc` is the probability that the player holding the gun successfully hits *and* you win: `p_suc[i][j] = 0` if $j = 0$ (if you shoot yourself, then you don't win) and `p_suc[i][j] = p * prob[i-1][j%i]` otherwise, where p is the current probability that a shot from the gun hits (equal to i/c).

Computing the values `prob[i][j]` isn't as easy as it seems, though, since each probability in this formula depends on itself – it's possible for the gun to come back to the person who now holds it without anybody losing. Generally, such problems can be solved using Gaussian elimination, but we won't need it this time.

Suppose we've computed `p_suc[i][]` already and need to find out `prob[i][]`. The states (i, j) are connected in one or more cycles which can be found in $O(n)$ time. For any such cycle c of l elements, we can start with $j = c[0]$ and apply the formula for `prob[i][j]` repeatedly l times to obtain

$$prob[i][c[0]] = \sum_{m=0}^{l-1} p\_suc[i][c[m]] \cdot (1-p)^m + prob[i][c[0]] \cdot (1-p)^l .$$

For $i > 0$, we have $1 - p < 1$, so we can compute the sum in $O(l)$ time and divide it by $1 - (1-p)^l$ to get $prob[i][c[0]]$; once we know that, the other values on the cycle can be computed easily.

The last question remaining is: how to compute the probabilities modulo $10^9 + 9$ and pick the largest one? One option is to work with fractions, but their denominators grow quickly. It's better to compute all probabilities in the same form as the output, replacing division with multiplication by the multiplicative inverse – Fermat's little theorem says $q^{-1} = q^{mod-2}$, which can be computed in $O(\log mod)$. We only need to compute them once for each cycle, which doesn't affect the complexity.

These numbers are hard to compare, though. How to pick the largest one? Well, nothing stops us from computing the probabilities in decimals too and picking the largest of them. The only situation where this trick would fail is when there would be multiple candidates too close to distinguish which is greatest (or if they're equal), but checking the data shows there's always only one maximum that's greater than all other probabilities by at least 10^{-10}.

Problem S: St. Ives

As I was approaching St. Ives,
Out came a man with seven wives,
Each wife had seven sacks,
Each sack had seven cats,
Each cat had seven kits:
Kits, cats, sacks, and wives,
How many were there going to St. Ives?

Problem specification

Above you see one of many versions of an old nursery rhyme. The **easy subproblem S1** is about a generalized version of the riddle contained in the nursery rhyme.

The nursery rhyme contains 5 types of objects: the man, the wives, the sacks, the cats, and the kits (i.e., kittens). In the generalized version there are n types of such objects. We will number these types 0 through $n-1$, in order.

In the nursery rhyme there is one man, and each object other than a kitten has 7 objects of the following type. In the generalized version there is one object of type 0, and each object of type i has a_i objects of type $i+1$.

In the **easy subproblem S1** you are given the number n and the numbers $a_0, \ldots, a_{n-2}$. Compute and output the answer to the riddle.

In the **hard subproblem S2** the input data is the same but the question you have to answer is different. The question will be revealed to you after you solve the subproblem S1. (More precisely, you will find the question for S2 in the evaluation details of an accepted submission to S1.)

Input specification

The first line of the input file contains an integer $t = 100$ 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 the positive integer n. The second line contains the nonnegative integers $a_0, \ldots, a_{n-2}$. You may assume that $1 \leq n \leq 10$ and that for each i, $0 \leq a_i \leq 10$. Note that for $n = 1$ the second line of a test case will be empty.

Output specification

For each test case output a single line with a single integer: the total number of objects that were, according to the riddle, going to St. Ives.

Example

input	output
1 4 0 10 2	1

In the example test case there is a man with zero wives, each wife has 10 sacks, and each sack has two cats.

Task authors

Problemsetter:	Michal ‘mišof’ Forišek
Task preparation:	Michal ‘mišof’ Forišek
Quality assurance:	Monika Steinová

Solution

There is a catch in the original riddle. Read the first two lines again:

As I was approaching St. Ives, out came a man with seven wives, ...

The man with his wives, sacks, and whatnot was not going to St. Ives at all – they were all *leaving* the town! Only you (i.e., the narrator) are going *to* St. Ives!

Thus, in the easy subproblem S1 the correct answer to any test case is 1. (You may have noted that we tried to disguise this by carefully chosing the example in the problem statement.)

Once you solved S1, you were informed that S2 is the problem you may have solved initially: count all objects that were leaving St. Ives. This is, obviously, $1 + a_0 + a_0a_1 + \cdots + a_0a_1 \ldots a_{n-2}$.

The constraints are small, hence this value always fits into an ordinary 32-bit integer.

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
```


Task authors

Problemsetter:	Miška 'Šandyna' Šandalová
Task preparation:	Michal 'mišof' Forišek, Mário Lipovský
Quality assurance:	Samko 'Hodobox' Gurský, Bui Truc Lam

Solution

Guessing the optimal solution for a square

A single isolated water cell does not spread at all.

With two water cells the best we can produce is obviously a 2×2 square: put the water cells diagonally adjacent to each other and wait for the water to spread.

After some experimentation it should be obvious that the best you can do with n water cells is an $n \times n$ square. Or, as seen from the other side, an optimal solution for an $n \times n$ square seems to involve n manually placed water cells.

In particular, one optimal solution for a square is to change one of its diagonals into water. The water will then spread to adjacent diagonals, eventually covering the entire square:

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

Of course, this is by far not the only optimal pattern that works. For example, a 4×4 square can also be optimally flooded like this:

```
.W..
W...
...W
..W.
```

Still, the task was to find any one optimal pattern, and the diagonal is almost certainly the easiest one to implement.

Proof of optimality

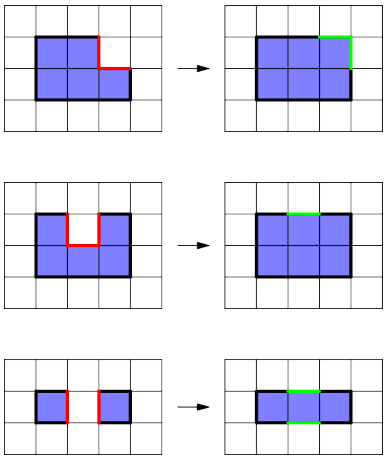
The cutest part of this problem is that there is a really pretty proof that the solution shown above is optimal. This proof does also work for rectangles.

The key observation is to look at the **circumference** of the flooded area. Remember that a cell gets auto-flooded if and only if at least two of its neighbors are already water cells. This means that whenever a cell gets flooded, the circumference of the flooded area *cannot increase*. If the newly flooded cell had two neighbors, the circumference stays the same, if it had more neighbors, the circumference decreases.

At the end of the automated flooding process the entire rectangle should be covered by water. The circumference of the rectangle is $2(r + c)$. Therefore, any valid set of manually selected cells must have a circumference of *at least* $2(r + c)$. And as each click increases the circumference of the flooded area by at most 4, we need at least $2(r + c)/4$ clicks.

If the rectangle is an $n \times n$ square, the above proof tells us that we need at least n clicks. Hence, any solution that uses exactly n clicks, including the one presented above, is indeed optimal.

Below are a few examples of how the circumference changes when the water spreads.



Dealing with rectangles

From the proof shown above we know that in the general case we need at least $\lceil (r+c)/2 \rceil$ clicks. All that remains is to show that patterns with this exact number of clicks always exist.

One possible construction looks as follows: Without loss of generality, let's assume that $r < c$ and that $c = r + d$. The minimum number of clicks can now be rewritten as $r + \lceil d/2 \rceil$. Thus, we can use r clicks to fill an $r \times r$ square, and another $\lceil d/2 \rceil$ clicks to fill the remaining d columns.

Here is one such pattern: even d on the left, odd d on the right.

```
W........       W.........
.W.......       .W........
..W......       ..W.......
...W.....       ...W......
....W.W.W       ....W.W.WW
```

We can easily verify that this pattern fills the entire rectangle with water, and from the proof we know that the number of clicks is optimal, q.e.d.

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

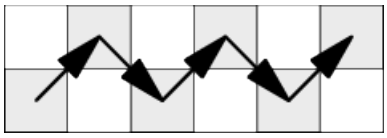
input

```
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".

Task authors

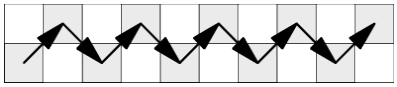
Problemsetter:	Michal ‘mišof’ Forišek
Task preparation:	Monika Steinová
Quality assurance:	Tomi Belan, Samko ‘Hodobox’ Gurský

Solution

Let’s assume that $r \leq c$. (Otherwise we solve the problem for swapped dimensions and transpose its solution.)

If $r = 1$ then bishop cannot move along the diagonals anywhere and thus the path can only consist of a single square. This is possible for the instance with $c = 2$ for white squares and $2 \leq c \leq 3$ for black squares.

All instances with $r = 2$ can be solved by a zig-zag pattern:

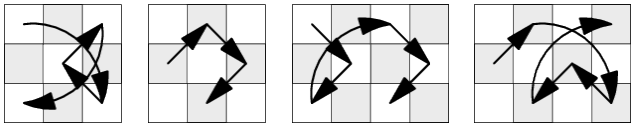


The solutions above work both in B1 and B2. Below we consider each subproblem separately for larger chessboards.

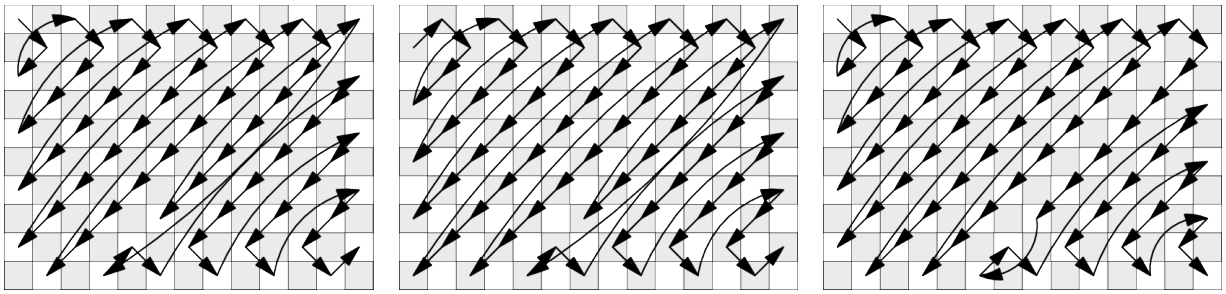
Easy subproblem B1

The ability to move the bishop along diagonals to any square enables a bishopian path in most instances.

For $r = 3$ one solution for each of the smallest few instances is shown on the following pictures. It is easy to generalize these patterns to chessboards with more columns.



There are some general patterns that essentially work for all larger chessboard dimensions, only with slight variations due to the parity of the dimensions and the chosen color. Few examples of such general patterns are shown below.



Hard subproblem B2

Once we introduce additional constraints on the bishopian path the problem changes substantially. Perhaps counterintuitively, most of the instances are impossible to solve.

First of all, let's realize that a non-intersecting bishopian path ("NI bishopian path" below) has another equivalent definition: it is a bishopian path in which the bishop always moves by a single square only. (A path that is non-intersecting cannot contain longer jumps. Each of the cells that has been jumped over has to be visited at some other point, and doing so will create an intersection. On the other hand, any bishopian path that only contains moves of length 1 is clearly a non-intersecting path.)

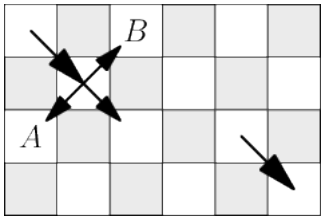
An instance on white squares with $r = 3$ and $c \geq r$ cannot contain a NI bishopian path. The squares $(1, 1)$ and $(3, 1)$ each have only one neighbor, so they have to be the beginning and the end of the path. However, they both have the same neighbor: square $(2, 2)$. This forces a path that consists of just these three cells and cannot be extended to other columns.

for the rest of this solution (and also in all the theorems) let's assume that $4 \leq r \leq c$.

Theorem 1. Let $4 \leq r \leq c$ be an instance of the problem. Then there exists no NI bishopian path on white squares.

Proof: In addition to the square $(1, 1)$ at least one other corner square of the chessboard is white. (Its location is determined by the parity of r and c.) Each of these corner squares is reachable only from a single white square and such squares thus have to be the endpoints of every NI bishopian path.

WLOG we can assume that the NI bishopian path will start in the top-left square $(1, 1)$. Then it necessarily has to continue to $(2, 2)$ from where we have three possibilities: $(1, 3)$, $(3, 3)$ and $(3, 1)$. Whichever square we pick there will remain at least one white square which will be reachable from a single white square. (In the figure below it is either square A or B.)



Hence we now have at least three white squares which are each reachable from a single white square. This concludes our proof: a path cannot have three endpoints and thus the NI bishopian path cannot exist.

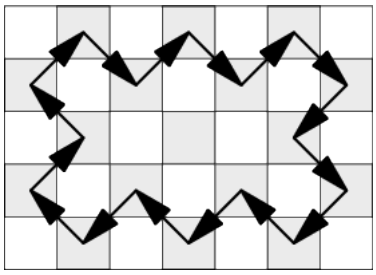
Theorem 2. There is no NI bishopian path on black squares in any instance where at least one of r and c is even.

Proof: It is easy to see that there are two black corners. We can swap the colors and rotate the chessboard so that the top left square is white. Theorem 1 now applies.

What remains to solve are the instances where both r and c odd, and we want a path on black squares.

Theorem 3. For any NI bishopian path on black squares with r and c odd: at least one endpoint of the path is located on the border of the chessboard. (I.e., there is an endpoint (x, y) such that $x \in \{1, r\}$ or $y \in \{1, c\}$.)

Proof: By contradiction. Let's assume that we have a NI bishopian path with no endpoint on the border The path has to enter and leave each square on the border of the chessboard at some point. Drawing these parts of the path results in a single closed path, as depicted on the following figure.



All edges on this cyclic path are forced, and therefore there cannot be any NI bishopian path – those don't contain cycles.

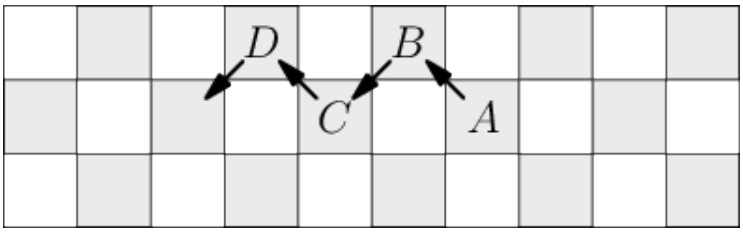
Theorem 4. For any instance on black squares with r and c odd, there exists a NI bishopian path in the $r \times c$ instance if and only if there exists a NI bishopian path in the instance $(r+4) \times (c+4)$.

We will first prove the direction from $r \times c$ to $(r+4) \times (c+4)$. From Theorem 3 we know that at least one of the endpoints of a NI bishopian path for the $r \times c$ board must be located on the border. This path can then be extended by a zig-zag pattern (as can be seen on figures below) to a NI bishopian path on the $(r+4) \times (c+4)$ chessboard.

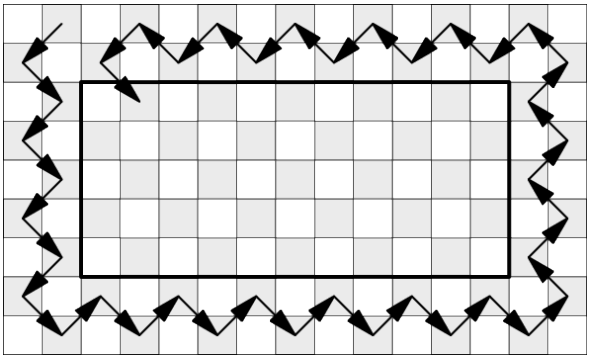
Now let's assume we have found a NI bishopian path on chessboard $(r+4) \times (c+4)$. We will discuss its properties which will enable us to shorten it to the instance $r \times c$.

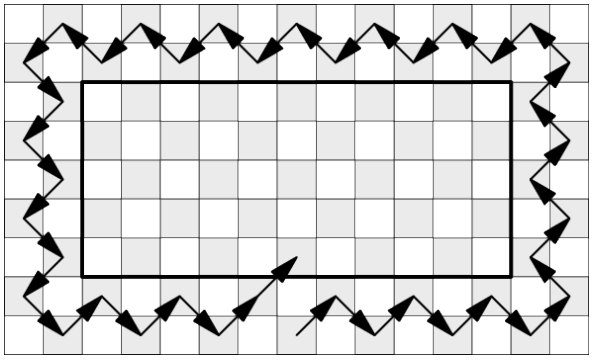
Let S_1 be the set of black squares at the border of the chessboard. Let S_2 be the set of black squares that would be at the border if we removed the set S_1.

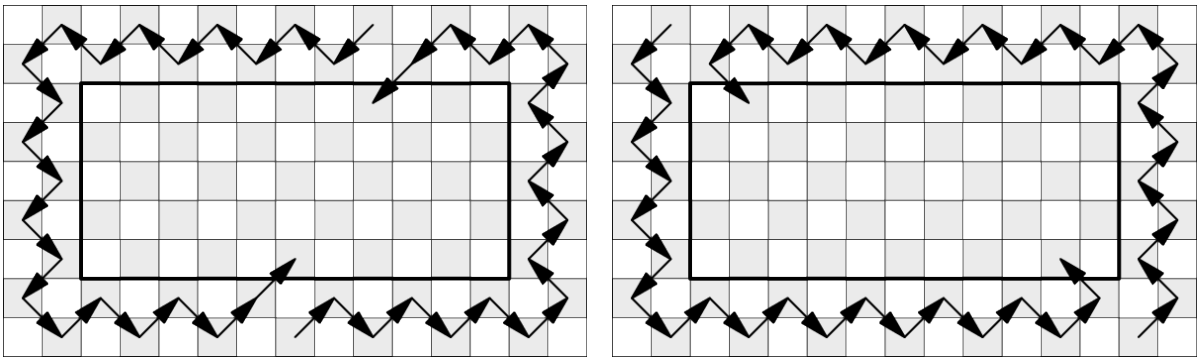
The set S_1 contains either one or both ends of the path. Except for these one or two squares, each other square of S_1 is an inner square of the path, which means that it's connected to both of its neighbors (squares in S_2). This induces a zig-zag pattern between squares of S_1 and S_2 as shown in the figure below.



The pattern will be formed around the entire chessboard, except for the one or two places where the path has an endpoint. The figures below show the main alternatives: one vs. two endpoints which may appear in a corner or on a side. (The case where one endpoint is in a corner and one is on a side may also occur but is ommited from these figures.)



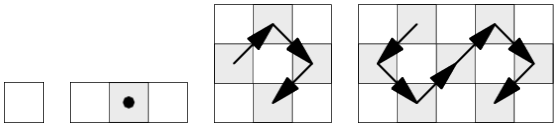




In all the cases the zig-zag pattern will eventually move from S_1 and S_2 to inner part of the chessboard with dimensions $r \times c$. As we can easily see, the remaining part of the original NI bishopian path will always be a NI bishopian path on the inner $r \times c$ part of the chessboard, which is precisely what we wanted to prove.

Theorem 5. For any instance on black squares with r and c odd, a solution exists if and only if $|r - c| \leq 2$.

Proof: We can use Theorem 4 to repeatedly shrink the rectangle until one of its dimensions becomes 1 or 3. The only rectangles with these dimensions that have a valid solution are shown in the figures below. (As a notable slightly-special case, when shrinking a 5×5 instance to a 1×1 instance, the entire path on the 5×5 instance lies on the boundary, it never enters the middle 1×1 area, as there are no black squares there. Feel free to check that a 5×5 instance does have a valid solution.)



Acknowledgement. Parts of the proofs shown above were adapted from a paper by G. R. Sanchis and N. Hundley.

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].

Task authors

Problemsetter:	Michal 'majk' Švagerka
Task preparation:	Michal 'majk' Švagerka
Quality assurance:	Michal 'mišof' Forišek

Solution

Expressing the probability

At the first glance this is a relatively simple task on Bayesian probability. Let X denote the event that we obtain the given multiset of emojis. We are asked to calculate the conditional probability $P(d \in [a,b]|X)$.

Using some simple algebra and Bayes' theorem, we can rewrite this probability as follows:

$$P(d \in [a,b]|X) = \sum_{q=a}^{b} P(d=q|X) = \frac{\sum_{q=a}^{b} P(X|d=q) \cdot P(d=q)}{P(X)} = \frac{\sum_{q=a}^{b} P(X|d=q) \cdot P(d=q)}{\sum_{q=1}^{m} P(X|d=q) \cdot P(d=q)}$$

How to find the probability of observing the sample, given the size q of the set from which we're sampling the emojis? We can use a combinatorial argument of counting the number of ordered samples that match the observation. Below, let $s = \sum x_i$.

First of all, we have to choose which specific emoji types to use. There are q ways of choosing the emoji type for x_1, $q-1$ ways for x_2, and so on. Once we do so, there are

$$C_x := C(x_1, x_2, \ldots, x_N) = \frac{s!}{\prod x_i!}$$

ways of ordering the individual emojis. However, if we just multiply these values, we are counting some orders multiple times. The formula would be correct if all x_i were distinct. However, if we have groups of identical x_i, we are counting each sequence multiple times. To correct for this, we have to divide the count by $D_x = \prod y_j!$, where y_j is the number of x_i that are equal to j.

Thus, the number of sequences of receiving s emojis in such a way that we get the given counts x_i can be expressed as $q(q-1)\cdots(q-n+1) \cdot C_x/D_x = q^{\underline{n}} \cdot C_x/D_x$.

The number of all sequences of receiving s emojis is simply q^s. Thus, the probability of observing the event X for a given number q of emoji types is

$$P(X|d=q) = \frac{q^{\underline{n}} \cdot C_x}{q^s \cdot D_x}$$

Easy subproblem

In the easy subproblem we have $P(d=q) = 1/m$ for each q between 1 and m, inclusive. Observing that neither this probability nor the values C_x, D_x depend on q we can cancel out these terms. This leaves us with the following simplified formula:

$$P(d \in [a,b]|X) = \frac{\sum_{q=a}^{b} (q^{\underline{n}} \ / \ q^s)}{\sum_{q=1}^{m} (q^{\underline{n}} \ / \ q^s)}$$

Since the limits are small, we can simply evaluate the expression.

Hard subproblem

In the hard subproblem there are a few complications. First, the probability distribution is different. If we unravel the definition, we'll see that it tells us that the probability of having $d = q$ is proportional to $1/q^2$. And as $\sum_{q\geq 1} 1/q^2 = \pi^2/6$, it must be the case that $P(d = q) = 6/(\pi q)^2$.

How does that affect the probability formula? As it turns out, not really. The constants again cancel each other out, and the q^2 just increases the exponent of q in the denominator.

A bigger complication is the fact that now the sum in the denominator has infinitely many terms. Can we perhaps find a closed-form solution of the sum?

We know that $P(X|d = q) \cdot P(d = q)$ will now simplify to the fraction $\left(q^{\underline{n}} \,/\, q^{s+2}\right)$. The numerator is a polynomial of order n. Its coefficients are the (signed) Stirling numbers of the first kind: $S_{n,k}$. Thus, we can rewrite this fraction as follows:

$$\frac{q^{\underline{n}}}{q^{s+2}} = \sum_{k=1}^{n} \frac{S_{n,k}}{q^{s+2-k}}$$

We can easily see that each of the fractions on the right hand side has the form "constant divided by q to the power 2 or more", from which we can easily show that if we take an infinite sum of all these fractions over all q, the sum will be absolutely convergent. Hence, we may rearrange the terms arbitrarily. In particular, we may group terms with the same Stirling number together. We get the following sum:

$$\sum_{q=1}^{\infty} \frac{q^{\underline{n}}}{q^{s+2}} = \sum_{k=1}^{n} S_{n,k} \cdot \sum_{q=1}^{\infty} \frac{1}{q^{s+2-k}} = \sum_{k=1}^{n} S_{n,k} \cdot \zeta(s+2-k)$$

It turns out that we **cannot** sum this analytically, as we don't know closed form solutions for Riemann zeta for odd positive integers. But these values are known to many decimal places – for instance the $\zeta(3)$ value (Apéry's constant) is evaluated to more than 10^{11} decimal digits. That's certainly much more than we'll need. We also know algorithms that can compute these values with an arbitrary precision.

The values we work with are rather large. For example, $S_{n,1} = (-1)^{n-1}(n-1)!$. This is a nightmare for numerical stability, and the final division only makes matters worse. We absolutely need more precision than standard floating-point numbers can offer. There are some tricks that we can employ to alleviate this issue, such as substracting 1 from each zeta value (which does not change the solution), giving us the ability to represent the values for large k (which are otherwise very close to 1), reordering the terms in the sum, but this is very hard to get right. A better solution is to use some implementation of big decimals, and to recompute the results with increasing precision until they become numerically stable. For example, we have a solution in Python using mpmath and a solution in Java that uses the built-in BigDecimal data type.

Solution summary

1. Obtain the values of the Riemann zeta function for small positive integers.
2. Use those to evaluate the denominator to several hundreds of bits of precision.
3. Evaluate the numerator as in the easy subproblem – summing it one term at a time.
4. Divide the two values to get the answer.

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
```


Task authors

Problemsetter:	Michal ‘mišof’ Forišek
Task preparation:	Michal ‘mišof’ Forišek, Tomi Belan
Quality assurance:	Tomi Belan, Michal ‘mišof’ Forišek

Solution

In this fun little problem you should have a bit of mathematical intuition, and you should trust it.

The easy subtask can be solved in various ways. For example, we can do a simple dynamic programming: for each n, what is the smallest k? In order to solve this problem for a particular n, try all valid choices of a_i, each will give you a smaller (i.e., previously solved) subproblem.

(See the published source code for more details.)

Large subproblem

Once we examine the correct output for the easy subproblem, we should see that there are only two types of n: if n itself isn't boring, the answer has $k = 1$ and $a_1 = n$, otherwise there always was an answer with $k = 2$.

Could this be true also for larger values of n?

Among the numbers up to $2 \cdot 10^9$ there is quite a lot of numbers that aren't boring. If we just limit ourselves to 9-digit numbers, there are $9 \times 9! = 3\,265\,920$ of those. If we take all pairs of such numbers, we will have about 10^{13} different pairs, each with a sum between 1 and $2 \cdot 10^9$. That's, on average, about 500 pairs per sum.

Now comes the part about trusting your intuition. The non-boring numbers are spread relatively well, so we can expect that the above is what actually happens in practice. From solving the subproblem D1 we also know that this is true for all small values of n.

Hence, we may expect that for the given range of n the answer is always either 1 or 2, and that there are always multiple pairs $a_1 + a_2$ that work.

With that in mind, we can write a very simple solution: if n is not boring output n, otherwise keep trying random a_1 until you hit one such that neither a_1 nor $n - a_1$ is boring.

Even if we don't trust our intuition, we have nothing to lose by trying this approach. If it finds a solution, we know that it is optimal. And if it doesn't, at least we'll learn some specific values of n that probably need $k > 2$.

Luckily for us, it turns out to be the case that the above simple solution actually works, and we are done with the task.

Extra credit

While preparing this task we verified that all numbers up to $2 \cdot 10^9$ have the above property. However, this has to break somewhere – after all, the numbers that are not boring have at most 10 digits. What is the smallest n for which we need $k = 3$?

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.

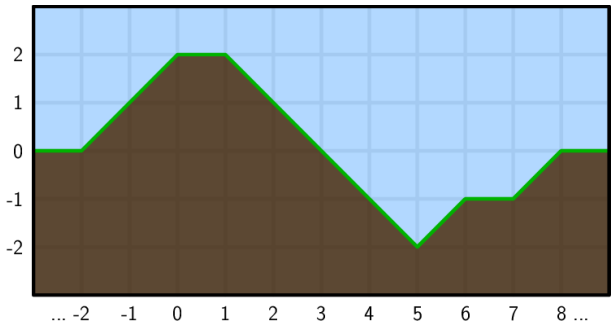
On a satellite image (i.e., looking from above), the future location of the railway appears as an infinite straight line. But you're more interested in the terrain's elevation profile (i.e., looking at the side view). Right now, everything is completely flat. That's boring! You need some more interesting 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

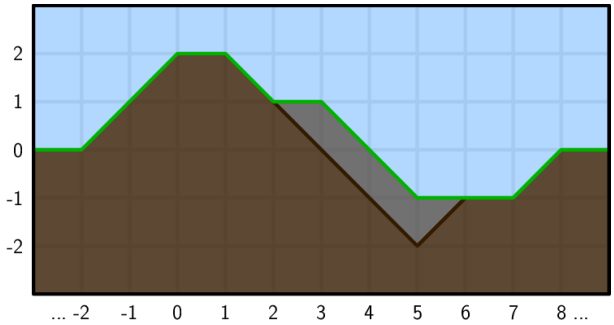
You took the satellite image and made a mark every 1 meter. You assigned those marked points coordinates from negative infinity to positive infinity.

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.

Task authors

Problemsetter:	Peter 'ppershing' Perešíni
Task preparation:	Tomi Belan
Quality assurance:	Michal 'majk' Švagerka

Solution

Let's consider what happens on every command. If we want to raise a point by 1 meter, and both its neighbors are on the same level or higher than it, then it's easy – we can just raise that point. If a neighbor is already lower than us, we must also raise that neighbor, and keep raising points in that direction until we find a neighbor that's equal or higher. For every command, we might have to change the elevation of many points.

But what if instead of remembering the elevation at every point, we just remember the slopes between neighbors? For example, instead of (10, 10, 11, 12, 12, 11, 10, 9, 8, 9) let's remember (0, 1, 1, 0, -1, -1, -1, -1, 1).

With this point of view, raising a point works like this:

- Find the nearest slope to the right of our point whose value is not -1. (This slope is between the last point we're raising and the closest point we don't have to raise.)
- Decrease that slope by 1.
- Find the nearest slope to the left of our point whose value is not 1.
- Increase that slope by 1.
- The cost can be calculated from the distance of the two slopes you changed.

Lowering a point is the opposite.

This can be done quickly by using a map (a.k.a. tree map, a.k.a. balanced search tree). Instead of storing all the slopes in an array, we can do "run length encoding" and store every continuous run of equal values as a single entry. This can be quickly queried and updated. The time complexity is $O(n \log n)$, because the terrain is initially flat and every command can change at most two slopes.

Problem F: Flipping and cutting

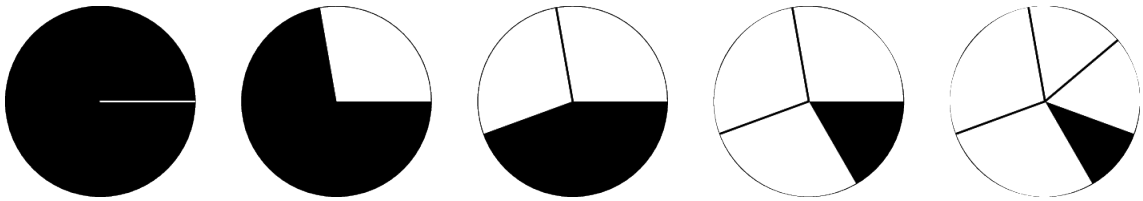
On the table we have a perfect circle cut out of paper. The circle is all black from the top and all white from the bottom. The circumference of the circle is a positive integer c.

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.

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'mišof' Forišek
Quality assurance:	Bui Truc Lam

Solution

This problem is based on a nice puzzle called "The Ice Cream Cake puzzle". Its origin is unknown, but you can find it, for example, in Peter Winkler's book "Mathematical Mind Benders".

The main interesting thing about this problem is that – contrary to what many people intuitively expect – the answer is *always* finite, even if the step happens to be irrational. Yes, even if we never cut the circle twice in the same place, it will still always return to the all-black state after finitely many steps. We will now show why that is the case and also how to compute the minimal number of steps needed.

How is that even possible?

The key observation is that the locations of cuts don't necessarily matter. After the cut, we always flip and reverse a part of the circle. If the reversal brings back a piece of the same color, the cut will, for all practical purposes, disappear – as if we never made it.

The only thing that matters are the coordinates at which a white piece touches a black piece. Below, we will call these coordinates *borders*.

Changing the game

Let's change the game slightly. The new game will be completely equivalent to the old one, but it will be a bit easier to argue about it.

First of all, let's scale everything. From now on, we'll assume that the step length x is an arbitrary real from the open interval $(0, 1)$, and that the circumference of the circle is exactly 1.[1]

The second change: instead of going around the circle, we will rotate the entire circle by x (measured along its circumference) after each flip. This way we'll get an equivalent process, but we will always be making the cut at the same place, and we will always be flipping a wedge that lies on the same part of the table.

Imagine a fixed coordinate system around the circle. (I.e., the coordinates don't rotate when the circle does, they are written on the table.) The coordinates wrap around the circle – i.e., all operations with them are done modulo 1. For example, the coordinates 0.47, 23.47 and -0.53 all represent the same point.

But back to our game. We will make the first cut at 0. Then, each round will consist of three steps:

- Rotate the entire circle by x counter-clockwise.
- Make a cut at 0.
- Flip the wedge that corresponds to the interval $(0, x)$.

A simple special case

The game is simple if the circumference is a positive multiple of x. If it happens to be the case that $1 = kx$, the game clearly ends after $2k$ steps: first we flip everything upside down in k steps and then we flip it back in another k steps.

[1] In the task statement we just use some special values of x, but the proof and the algorithm work for all positive reals.

The remaining general case

Below, we will deal with the remaining case. Let $k = \lceil 1/x \rceil$.

If we start with an all-black circle, the first $k-1$ cuts will flip a black wedge to its white side. Then, the k-th cut will cut into the first wedge.

Let's write $x = y + z$, where $y = 1/k$. After the first round we have two borders: at 0 and at x. After the second round the borders are at 0 and $2x$. After $k-1$ rounds the borders are at 0 and $(k-1)x$. In the following round we get a new border at $x - kz$. In the following rounds new borders appear at $2x - kz$, $3x - kz$, ..., $(k-1)x - kz$. And that's it. We can easily show that if the current borders are only at some of these coordinates before a round, there will only be borders at some of these coordinates (and nowhere else) after the round.

Finiteness of the game

The current state can be described by specifying the current set of borders and the color of one of the pieces between them. As there are only finitely many possible locations of borders, a state must eventually repeat. As the process is deterministic, it must be eventually periodic. And as the process is reversible, it must be purely periodic. Hence, the first state to repeat must be the initial state when the circle is all black.

Period length

The full set of border lines divides the circle into $2k-1$ areas. Out of those, k have length $(x - kz)$ each, and between them (except for zero) $k-1$ areas have length $4z$. Let's call the first ones "areas of type A" and the second ones "areas of type B".

It is easy to show that areas of type A change their color periodically. As we do the rounds of our process, first we turn them from black to white, one at a time, and then we turn them back from white to black, one at a time. Thus, all areas of type A are black only once every $2k$ rounds.

The same holds for areas of type B. But as there is only $k-1$ of those, they are only all black once every $2(k-1)$ rounds.

It follows that the entire circle is black once every $2k(k-1)$ rounds. (Note that k and $k-1$ are relatively prime.)

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
```


Task authors

Problemsetter:	Tomi Belan
Task preparation:	Tomi Belan
Quality assurance:	Jakub 'Xellos' Šafin

Solution

Easy subproblem G1

The characters in the film form a forest. The queries are to change the parent of a vertex and to find out if one vertex is an ancestor of another.

Consider the string printed by tree traversal. For example, traversing a tree of three vertexes where 2 and 3 are children of 1 would print "entering 1, entering 2, leaving 2, entering 3, leaving 3, leaving 1". Strings like this are very useful for this problem.

If we could quickly find the position of a given entry in the string, we would be able to answer whether b is an ancestor of a by checking whether "entering a" is between "entering b" and "leaving b".

And if we could quickly split and concatenate those strings, we could change the parent of a vertex by finding the substring from "entering a" to "leaving a" (inclusive), cutting it out of its current place, and inserting it into a new location.

We can do this with a balanced binary tree, such as a treap. Each node of the treap will be either "entering x" or "leaving x", and their order will indirectly represent the structure of our original tree. By keeping a reference to every node and remembering the size of all subtrees of the treap, we can quickly find out the index of any treap node by counting how many nodes are to the left of it.

Note that we don't use the treap as a binary *search* tree – the entries aren't ordered by keys, but explicitly by split/merge calls.

Hard subproblem G2

In the hard subproblem, we just need to add support for cycles. We keep the above structure, but for every tree in the forest, we also remember an optional "phantom" edge from the root vertex to a vertex in the same tree. This edge won't be stored in our special structure, but in a plain array.

For simplicity, instead of "changing the parent of vertex a to vertex b" let's the query into two parts: "removing the edge from a to its parent" and "adding a parent edge from a (which is currently a root) to b".

To add an edge from a to b:

- If a is already an ancestor of b, just add a phantom edge from a to b.
- Otherwise it's the case without cycles: take the traversal string representing a and insert it right after "entering b" (or right before "leaving b").

To remove an edge from a to its parent:

- If it's the phantom edge: just delete it.
- Find the root of this forest (r) and check if it has a phantom edge to some other node c.
- If yes, that means there is a cycle between r and c. Check if a is on that cycle (a is an ancestor of c).

- If a is on the cycle, then by breaking the edge from a to its parent, a will become the new root. Remove the phantom edge from r to c, and replace it with a normal edge from r to c. Now a will be the root and r will be its descendant.
- Otherwise, there is no cycle, so cut out the traversal string representing a and make it a new root, as usual.

The amortized time complexity is $O(n + q \log n)$.

Behind the scenes

It's always difficult to prepare input data for problems like this, where a $O(n\sqrt{n})$ solution also exists and is often much easier to write. In a normal contest, if the optimal solution takes 1 second and a suboptimal solution takes 30 seconds, it's easy to distinguish them. But IPSC is an open data contest. If the optimal solution takes 1 minute and a suboptimal solution takes 30 minutes, it can still be worth it.

On one hand, we'd like to prevent $\sqrt{n}$ solutions, or at least make them slow enough to discourage most of them. On the other hand, making the input bigger also makes the optimal solution slower, and we don't want to disadvantage teams with slower CPUs or smaller RAM.

With the inputs we've chosen, our optimal solution takes about 4 minutes and a $\sqrt{n}$ solution takes about 80 minutes. Maybe some teams will give up on their optimal program and kill it after it runs for ten minutes without success, and maybe some teams will find a fast-enough suboptimal solution and save some developer time, but on average we hope it's a good compromise.

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.

Task authors

Problemsetter:	Lukáš Poláček and Jano Hozza
Task preparation:	Lukáš Poláček
Quality assurance:	Tomi Belan

Solution

Easy subproblem H1

In the easy subproblem, we can have multiple commands on one line separated by semicolons. The stack is initially empty but can stay non-empty when the program ends, so we can use the `push` command to determine the line order after sorting. The pushed values are ignored otherwise. For example, the following structure works:

```
push 1;... subroutine for x -> x ...;jump 3
push 0;... subroutine for x -> -x ...;jump 3
push 2;... subroutine for x -> x * x ...;jump 3
```

In the normal case, the subroutine for x is executed and then the execution jumps to the line number 3, which means the program exits. In the reversed case, the subroutine for x^2 is executed. Finally, in the sorted case the second line becomes the first and the subroutine for $-x$ is executed. The subroutines are respectively:

```
input;print
input;push -1;*;print
input;dup;*;print
```

The full program then becomes:

```
push 1;input;print;jump 3
push 0;input;push -1;*;print;jump 3
push 2;input;dup;*;print;jump 3
```

Hard subproblem H1

Arithmetic operations come before letters in the ASCII table. Also the `dup` command comes before all other word commands, so arithmetic operations together with `dup` are always first in the sorted order. However, all these operations crash with an empty stack and we don't get a chance to push anything onto the top of the stack before we execute them, so we cannot have lines starting with arithmetic operations or `dup`.

A trick to execute such operations is to wrap them in "`push ?`" and `pop`. E.g. for `+` we have:

```
push 42
pop;+
```

The general idea of the solution is to have 3 areas of the program separated by "`jump 99`" commands (assuming the program has 99 lines):


```
... subroutine for x -> x ...
...;jump 99
... extra lines for x -> -x ...
...;jump 99
... subroutine for x -> x * x when reversed ...
```

Let's label these 3 areas α, β and γ respectively. For γ, we can use the following sequence of commands (showed in reversed order for clarity).

```
input;dup
push ?
pop;*
print;jump 99
```

If there are multiple `input` commands in the program, we need to make sure only one gets executed each time. One line can start with `input` and another can contain "`push ?;input`". A `jump` command between them makes sure only one of them gets executed. As "`input;dup`" is already in γ, we use "`push ?;input`" in α. The whole α then becomes:

```
push ?;input
print;jump 99
```

We know that "`input;dup`" ends up at the beginning of the sorted program. A `jump` command comes lexicographically right after `input`, so we can add it to β and use it to jump to the part calculating $-x$ in the sorted program. The jump can go to a line starting with `pop`, `print` or `push`. Jumping to a `print` would be wrong, since we have x at the top of the stack at that point. Instead, we can calculate $-x$ by pushing -1 onto the stack and then multiplying the two top-most elements. So we point `jump` to the line "`push -1;*`", which we add to β.

The lines that follow in the sorted order after "`push -1;*`" start with `push` commands, which don't allow us to print $-x$ that is at the top of the stack, we jump back in the program using "`push z;jump` ℓ", where ℓ is a line higher up in the program. We just pushed z to the stack, so the line we jump to starts with `pop`. We want the line that comes after popping z from stack to be "`print;jump 99`", so the line popping z must come lexicographically after "`pop;*`" (from γ). We achieve it by using "`pop;jump` $\ell + 1$", which effectively continues on the next line.

To recapitulate, we have the following lines in β:

```
jump L         # Line number to be determined
push -1;*
push ?;jump K  # Line number to be determined
pop;jump K+1
```

Putting all pieces together, the program looks as follows (with line numbers for easier orientation).

```
00: push 2;input
01: print;jump 10
02: jump 6
03: push -1;*
04: push 1;jump 3
05: pop;jump 4
06: print;jump 10
```



```
07: pop;*
08: push 3
09: input;dup
```

The sorted program is the following.

```
00: input;dup
01: jump 6
02: pop;*
03: pop;jump 4
04: print;jump 10
05: print;jump 10
06: push -1;*
07: push 1;jump 3
08: push 2;input
09: push 3
```


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.

Task authors

Problemsetter:	Tomi Belan
Task preparation:	Tomi Belan
Quality assurance:	Samko 'Hodobox' Gurský

Solution

Can you believe we didn't think of that problem name until 2017?

For a server s to be hacked:

1. there must be a path from 1 to s,
2. there must be a path from s to n,
3. there can't be a path from s to s,
4. there can't be a path from 1 to n which skips s.

In the easy subproblem, we can use DFS or BFS to check all conditions directly for every server. E.g. to check condition 4, we can literally temporarily remove s from the graph and see if a DFS from 1 visits n. This is also useful for checking that your hard solution didn't miss any special cases.

In the hard subproblem, we need something faster.

First, we filter out all servers that aren't reachable from 1 or that can't reach n. This takes care of conditions 1 and 2, and makes sure that the other servers won't cause false positives for condition 4.

Then, we find the strongly connected components and a topological sort of the directed acyclic graph formed by them. Tarjan's algorithm can do both together.

For a server to fulfill condition 3, it must be the only server in its strongly connected component, and it also can't have a self-loop (s, s).

The final condition can be checked by looking at the directed acyclic graph and walking through the topological sort from 1 to n. (Recall that we removed all servers that can't be reached from 1 or that can't reach n.) During this, we remember the farthest edge we've seen so far. If there is an edge that starts from somewhere before the current vertex, and ends somewhere after the current vertex, then it's possible to use it to skip over this vertex and so it doesn't meet condition 4. If all edges from previous vertexes end in the current vertex or earlier, then it fulfills the condition.

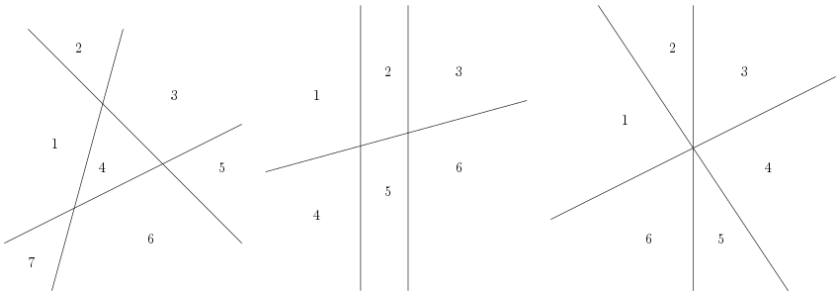
Problem J: Judicious cuts

"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.

Task authors

Problemsetter:	Michal ‘mišof’ Forišek
Task preparation:	Michal ‘majk’ Švagerka
Quality assurance:	Lukáš Poláček, Edo ‘Baklažán’ Batmendijn

Solution

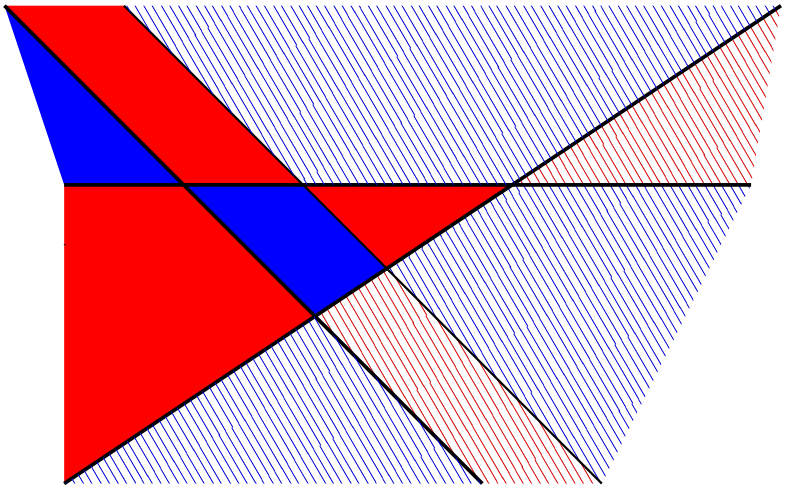
The easy subproblem is really really easy. To the point that you'll probably bang your head against the wall if you didn't get it. The easiest solution: output $r - 1$ distinct horizontal lines.

We will now take a look at the hard subproblem.

First let's look at how to count the number of regions for a given arrangement of lines. Our output format already ensures that there are no lines parallel to the y-axis, but in general, one can slightly rotate the plane to avoid those anyway. Consider any finite amount of non-vertical lines. We can now find the rightmost point of every region. There are two types of regions:

- Those that do not have a rightmost point, as they are **unbounded** on the right side. Since there are no lines parallel to the y-axis, it is easy to observe that there are exactly $\ell + 1$ of these.
- Those that do have a rightmost point. Such rightmost point always lies at the intersection of two or more lines. We call these regions **bounded**.

In the figure below, the regions filled with stripes do not have a rightmost point – those are the unbounded regions. The other five filled with solid colour are bounded.



When the lines are in a general position (that is, no three lines meet at a point and no two lines are parallel) then every pair of lines has a unique intersection point. Each intersection point serves as a rightmost point to exactly one region. Since there are $\binom{\ell}{2}$ such intersections, the total number of regions is

$$\frac{\ell^2 + \ell + 2}{2}.$$

This is also the maximal amount of regions one can make using ℓ lines. How do parallel lines and multiple lines meeting at a point affect the result?

The case of parallel lines is simple – if there is a set of k parallel lines, we have $\binom{k}{2}$ fewer intersection points which exactly corresponds to the number of regions we lose.

Let's look at m lines meeting at a point and ignore all the others for a while. If the lines did not meet at a single point, but they were in general position instead, there would be $\binom{m}{2}$ regions with a rightmost point. Now we have 1 intersection point that borders $2m$ regions. We already know that $m+1$ of them are unbounded, which leaves us with $m-1$ bounded regions. This means that the effect of an arrangement of m lines meeting in a single point on the number of regions is a decrease by $\binom{m}{2} - (m-1) = \binom{m-1}{2}$.

The above observations can now be used to count the number of regions for any configuration of lines. (This is sometimes called Roberts' formula.)

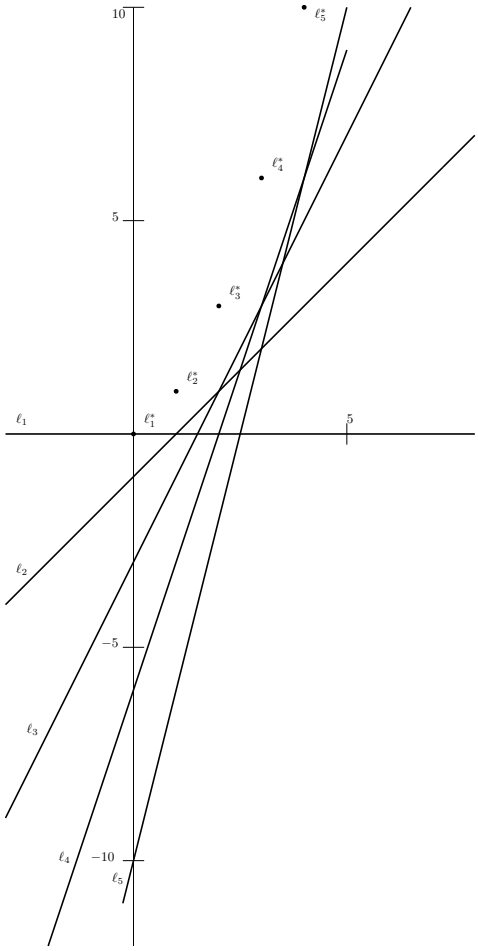
This gives us an idea how to construct the solution for a given r. If there is an ℓ such that $(\ell^2+\ell+2)/2 = r$, output ℓ lines in a general position. Otherwise, find the smallest ℓ for which the general position has more regions and try to remove some of them by introducing the "flaws" mentioned above. For example, if $r = 13$, we can take $\ell = 5$ lines. If they were in a general position, we would have 16 regions. If we instead place them so that four of them meet at the same point, this will reduce the number of regions by $\binom{4-1}{2} = 3$, leaving us with 13 regions.

There are many different constructions, for instance one can introduce some sets of parallel lines, and ensure that they do not create points where three or more lines intersect. The solution presented below uses the notion of a point-line duality and is able to stay well within the limits on line parameters.

Formally a **dual point** to the line L given by $y = mx + b$ is the point $L^* = (m, -b)$. Every line not parallel to y-axis has a dual point, and every point has a corresponding dual line. Duality has many interesting properties that are often useful in similar constructions. We use two of them here without proof:

- Two lines L_1 and L_2 are parallel iff their duals L_1^* and L_2^* have the same x coordinate.
- Three lines L_1, L_2, L_3 meet at a single point iff their dual points L_1^*, L_2^*, L_3^* are collinear. The intersection point is the primal of the line connecting the dual points.

We can imagine the construction, and also prove its properties easier in the dual representation. In the construction we avoid parallel lines completely, by setting m_i (the slope of the i-th line) equal to i. We let the y-intercept increase by i between consecutive points. E.g., for five dual points we have $\{(0,0),(1,1),(2,3),(3,6),(4,10)\}$ (depicted in the following figure). What this construction effectively does is increasing the slope of the dual line between x consecutive dual points. It is easy to prove that no three such dual points lie on a dual line: since the slopes of lines connecting different pairs of dual points are different, so is the x-coordinate of intersection of their primals. When the intersection points have different x-coordinate, they **are** different points. Hence, no three (primal) lines meet at a (primal) point and this approach gives us ℓ lines in general position.



How to decrease the number of regions? Let's start by always outputting a primal line $(0, 0)$. Let x denote the number by which we need to decrease the number of regions to achieve the desired amount. Once we reduce x to zero, we are done. In order to do that, we pick the largest c such that $\binom{c-1}{2} \leq x$. Introducing c collinear dual points reduces x to $x' = x - \binom{c-1}{2}$. The simplest and efficient way to introduce c collinear points is to add $c - 1$ points collinear with the last. We do this by keeping the same slope for all. For example, the last inserted point is $(4, 4)$ and we demand 3 consecutive points. We use slope 5 for both, that is we output points/lines $(5, 9)$, $(6, 14)$. The point $(6, 14)$ is now the last point, and we can use it for another set of collinear points (e.g., $(7, 21)$, $(8, 28)$).

The thing left to do is to assess whether this construction is efficient enough. As introducing k collinear dual points removes $\Theta(k^2)$ regions, and adding an extra line gives us $\Theta(\ell)$ new regions, the intuition is that for large number of lines this construction is sufficient. A bit of case-work reveals the construction does **not** work for $r \in (3, 5, 9, 12, 17, 24)$, where one would aim to use $(2, 3, 4, 5, 6, 7)$ lines, respectively.

In all cases but $r = 5$ and $r = 17$ one can fix the construction by using one set of parallel lines at the end (k parallel lines are sufficient to reduce r by $\binom{k}{2}$, whereas $k+1$ intersecting lines are needed; and we can again reuse the last inserted point[2]), or by hand-crafting the solutions. Case $r = 17$ can indeed

[2]Notice that when using just parallel lines, one cannot reuse the last point for the next set, as parallelity is transitive. It is also harder to prevent intersecting lines.

be done using 6 lines, but even the more efficient construction fails to find the solution, we're left with solution by hand.[3]

Finally, we're left with $r = 5$. This is indeed a special case where we need one more line than the lower bound we established before. We're left with no choice – the approach used for the easy subproblem is optimal here.

In the scope of the contest, it is advisable to leave the proof out, follow the intuition and then solve by hand the few cases where the construction failed. If we're lazy, we can drop the negative sign by y-intercept (this corresponds to reflection of the dual along x-axis, which definitely doesn't affect the solution and the properties of the duality we're using).

Bonus question: What if parallel lines were not allowed?

[3] Hint: one set of 2 parallel lines, one set of 3 parallel lines, and a 3-way intersection.

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.

Task authors

Problemsetter:	Askar Gafurov
Task preparation:	Michal 'mišof' Forišek
Quality assurance:	Tomi Belan

Solution

Linear congruential generators (LCGs) are one of the most popular sources of pseudorandom numbers. The main reasons why this is the case: they are very fast and simple. However, the quality of the random numbers they provide isn't really high.

Additionally, not all LCGs are equal. By its nature, each LCG is periodic, but different LCGs can have periods of different lengths and with different patterns. Some of them will pass at least some basic statistical tests, some won't.

In this problem we used two LCGs that aren't really among the best ones.

The first sequence was generated by a distorted version of the LCG used in Java's java.util.Random, POSIX [ln]rand48, and glibc [ln]rand48[_r]. This generator uses $m = 2^{48} - 1$, $a = 5\text{DEECE66D}_{16}$, and $c = 11$. Its official implementations output the 32 highest-order bits of the internal state (bit 47 down to bit 16 of the current remainder modulo m).

In our implementation we distorted this LCG: instead of outputting the highest-order bits (which have a large period and behave nicely) we output the 24 lowest-order bits.

The second LCG is the generator used in random0 to generate random floating-point values between 0 and 1. This generator has $m = 134\,456$, $a = 8\,121$, and $c = 28\,411$. The original generator outputs the value s/m after each step, where s is the current state. Again, our generator is actually a distorted form of this LCG: instead of doing this division, we simply output bits 16 down to 0 of the current state.

In the second LCG it should be quite obvious that there are multiple significant issues. First of all, the modulus is not a power of 2, which is why bit 16 of the current state is more likely to be 0 than 1. On the other end of the state we can observe an even simpler pattern: if the current state is odd, the next one will be even and vice versa. That should be pretty easy to observe and then to predict :)

There are many ways to solve this task. The correct submissions included both ones that barely made it over the 2500 correct per sequence, but there was also a lot of submissions that got something like 3998 + 3708 bits correct.

A painful but reasonable way of solving this task is based on statistical tests. Code up a bunch of tests, find irregularities, and use those to get good predictions for the missing bits.

One particular test that is worth looking at: for each pair of small integers $0 \leq q < p$ take all bits on positions $pi + q$ and for all small k count the number of occurrences of each k-bit substring.

Our favorite way of solving this task is to realize that there are some simple dependencies between adjacent bits. We don't know what they are, but we don't have to. We can just take some standard classifier (e.g., something from python-sklearn), train it using the known parts of the sequence ("in this context expect this bit") and then use it to predict the missing bits from their context.

(You can easily verify whether this approach works for a particular classifier by dividing the training data you have into training and validation data. In other words, train on 90% of data you have, verify that it predicts the remaining 10% well, and then use the trained classifier to fill in the actual blanks.)

We are looking forward to hearing from you: what method did you use? How painful was it?

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.

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'mišof' Forišek
Quality assurance:	Tomi Belan

Solution

This problem introduced our version of Penney's Game. An interesting feature of this game is that it is nontransitive. What does that mean? Intuitively, one might expect that there is an ordering of all possible patterns from the "best" one to the "worst" one, and that the optimal strategy for Yvonne is to simply pick the "best" pattern. This is not the case. Starting from $k = 3$, for any pattern there are other patterns that beat it – in other words, regardless of what Yvonne choses, Zara can always choose a pattern that is designed to beat Yvonne's specific pattern. Thus, in Penney's game for $k \geq 3$ Zara wins more than 50 percent of all games if she plays optimally.

Of course, our version of the game mixes stuff up a bit. The main differences between our game and the original: First, our game is finite: we will run out of cards eventually. Second, choosing a sequence removes some cards from the deck, which influences the probabilities.

The example annotation in the problem statement does already contain an important hint: a good strategy for Zara. Regardless of what sequence Yvonne chooses, Zara always has the option to choose the complement of that sequence. (The cards to do that have to be available – do you see why?) And if she does so, she is guaranteed a winning probability of exactly 0.5. Thus, for any valid k there are only two possibilities – either the game is a tie, or Zara has an even better strategy that tilts the game in her favor.

We already know what happens for small k. For $k = 1$ and $k = 2$ the game is a tie. For $k = 3$ in the original game Zara wins 2/3 of all games if both girls play optimally, and we may expect a very similar result here – after both girls choose their sequences, there are still many cards left in the deck and the ratio of reds to blacks is roughly 1:1, so this won't skew the probability by a lot. Thus, we may expect that for smallish k Zara should win.

What happens for largeish values of k? The easy thing to note is $k \geq 18$. Already for $k = 18$ after both girls choose their sequences there will only be 16 cards left in the deck, which isn't enough to construct either of the two chosen sequences. Thus, for $k \geq 18$ the game will surely end with a coin flip, and therefore it is a tie.

IF you got this far, this was a good place to solve the subproblem L1 by taking an educated guess that Zara wins for $3 \leq k \leq 17$. And if this doesn't work, we can always start from $k = 17$ downwards, change Zara's win to a tie and resubmit. This strategy would indeed solve L1, albeit with one bad submission. It turns out that the case $k = 17$ is still a tie.

Looking at 17-card sequences

We will now show that $k = 17$ is still a tie if both girls play optimally. One of many optimal choices for Yvonne is the sequence "9× red, then 8× black".

Let's examine Zara's options:

- If she chooses any sequence with 9 or more red cards, there will be too few red cards left in the deck for either sequence to appear. Hence, the coin flip will decide.
- If she chooses any sequence with 10 or more black cards, there will be too few black cards left in the deck for Zara's sequence to appear, so her probability of winning has to be at most 50%.

- The only remaining option is some sequence with 8 red and 9 black cards. What happens then? The deck will contain 9 red and 9 black cards, so there are $\binom{18}{9}$ equally likely possible orders of the deck. Two of these contain an occurrence of Yvonne's sequence and two contain an occurrence of Zara's sequence. If those four deck orders are all distinct, the situation is clearly symmetric and each girl wins with probability 50%.
- For some pairs (Yvonne's sequence, Zara's sequence) it will be the case that some order of the 18 cards in the deck will contain both sequences – cards #1 through #17 will form one of the sequences, cards #2 through #18 will form the other sequence. In that case, the winner will be the girl whose sequence appears sooner, and this will skew the overall probability in her favor.

For Yvonne's sequence we mentioned above (9× red, then 8× black) this can happen in only one way: the way where Yvonne wins. (If you place Yvonne's sequence as cards #2 through #18 in the deck, there will still be 9 red cards among the first 17, and therefore it cannot be Zara's sequence.)

Hence, if Yvonne chooses the sequence mentioned above, Zara only has a choice between some patterns that win her the game with probability 50% and some patterns that are even worse than that. And this means that (assuming both girls play optimally) the game is a tie for $k = 17$.

Computing the result of a single game

OK, that was as far as we can reasonably go without writing any code. Let's now take a look at evaluating a single game. That is, we ask the following question: given k and the patterns chosen by Yvonne and Zara, what is the probability that Yvonne wins the game (assuming optimal play)?

How can we answer this question? Obviously, by going through all possible deck orders one at a time. There are $52 - 2k$ cards in the deck, so there are roughly 2^{52-2k} possible deck orders. The actual number is a bit smaller, as the number of red cards in the deck is fixed, but this is still a useful estimate. Anything close to 2^{46} for $k = 3$ is more than we can reasonably afford.

How can we answer the question we have *in an efficient way*? By realizing that during those 2^{52-2k} many situations are often repeated, and therefore we can use *dynamic programming* to speed up the previous solution. One possible observation is that during the game all we need to know are the last k cards that were already played (or fewer if the game is just starting) and the number of red and black cards left in the deck. This gives us only something like $2^k \cdot 26^2$ states to examine – which is much better for small k.

However, there is also a much more compact way to represent the state. For example, imagine that Yvonne has the sequence RBRB and Zara has the sequence BBRR. Assume that the last seven cards played from the deck were RRRBBRB. What do we really need to remember about this sequence? Yvonne doesn't care about the first five of those cards, as those clearly won't help her form an occurrence of her pattern. The only thing Yvonne currently cares about is that the last two cards (RB) can be extended to form an occurrence of her pattern. Similarly, Zara doesn't care about the first six cards played from the deck – only the last B can be used by her.

This observation brings us to a much more compact representation of the state during a game: in addition to the number of red and black cards left in the deck, all we need to remember is the *longest suffix* of the sequence of already played cards that is *also a prefix* of one of the girls' sequences.

And as there are at most $2k + 1$ distinct prefixes (we only have two patterns, each of length k), this leaves us with at most $(2k + 1) \cdot 26^2$ different states of the game – which is very small for any valid value of k. Thus, we can now take any two patterns of red and black cards and very quickly compute the probability that Yvonne wins.

In order to compute this probability as an exact fraction, just note that we can represent it as (the number of games Yvonne wins) / (the number of possible deck orders × the number of possible coin

flips). For a deck with r red and b black cards the denominator is simply $2(r+b)!$. We can then use the dynamic programming described above to compute the numerator of this fraction.

Finding the result of optimal gameplay

In order to find the result if both girls play optimally, we need to evaluate a simple minimax decision tree. This can be expressed in pseudocode as follows:

```
best_result_yvonne = 0
for each possible pattern Yvonne can choose:
    best_result_zara = 1
    for each possible pattern Zara can now choose:
        pp = probability that Yvonne wins for those two patterns
        best_result_zara = min( best_result_zara, pp )

    best_result_yvonne = max( best_result_yvonne, best_result_zara )

return best_result_yvonne
```

In words: Given a particular choice by Yvonne, Zara always chooses the sequence that minimizes the probability that Yvonne wins. Given this information, Yvonne can evaluate each of her options and choose the one that maximizes her winning probability.

This can still be a lot of work: for example, for $k = 13$ there are exactly 2^{26} games to evaluate, and this is still not the largest case. Thus, we need to optimize this search somehow in order to speed it up.

In our solution we do two improvements: *pruning* and *reordering*.

As for pruning, we do the following: For Zara we never look at patterns for which she cannot win (as there will be too few cards of a color left in the deck), as these leave Yvonne with at least 50% win probability.

As for reordering, we do two things. First, we keep all of Yvonne's options in a priority queue. For each of them we remember the current winning probability and an index that tells us how many of Zara's answers we already examined. Obviously, the priority queue is ordered by winning probability. In each step, we take the top option (which is the current candidate for being the best answer) and try another of Zara's answers. When we do the search in this way, we can terminate it as soon as we finish testing the option that currently sits on the top of the priority queue.

The second improvement by reordering is that we "apply pretests": in order to speed up the search, we start the processing of each of Yvonne's patterns by testing it against a few possible options for Zara for which we guess that they may be good answers to this particular choice made by Yvonne.

In the infinite version of Penney's Game the best answer for Zara is usually a sequence such that its suffix of length $k-1$ is a prefix of Yvonne's sequence – in other words, Zara chooses a sequence that can appear immediately before Yvonne's sequence. Our version is distorted, but it turns out that Zara's sequences that end in a prefix of Yvonne's sequence are still among the best choices, so we'll start by testing some such sequences first.

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.

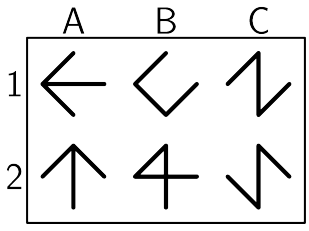
Output specification

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? :)

Task authors

Problemsetter:	Michal ‘mišof’ Forišek (also for the WPC puzzle)
Task preparation:	Michal ‘mišof’ Forišek, Tomi Belan
Quality assurance:	Tomi Belan, Michal ‘mišof’ Forišek

Solution

If you were to try to solve a puzzle like this by hand, you cannot afford a quadratic-time approach – comparing each pair of figures manually is too slow. Essentially, you need to come up with some good hashing function.

When looking for identical shapes and/or mirror images, distinct features of some shapes (e.g., vertices of degree 1) can be used to eliminate some of the figures quickly.

For a more robust hash function, take another few steps towards the complete (circular) sequence of vertex degrees. This is a natural extension of the previous approach, as you can do it incrementally. For example, once you are done with degree-1 vertices, you can look at the number and placement of degree-2 vertices as well, and that may already be enough.

This can also be generalized to looking for complements, but it’s much more painful. If I absolutely had to find the pair of complementary figures manually, I would probably start by spending a linear amount of time on writing the complete degree sequence around each figure. When looking for a complement of a particular figure, you then need to invert its sequence and recall/check whether you have the new one somewhere.

Of course, this being IPSC, you also had other options. In particular, you could write a program that parses the input image, recovers the individual graphs and finds the matching pairs by comparing each pair of graphs.

Probably the fastest way to solve this task would be a combination of both approaches: quickly write a simple program that can, for example, guess vertex degrees (by hard-wiring their coordinates and counting dark pixels around them) and use those to print a small set of candidate pairs to be examined by hand.

