Problem A. ASCII Art

Input file: `ascii.in`
Output file: `ascii.out`

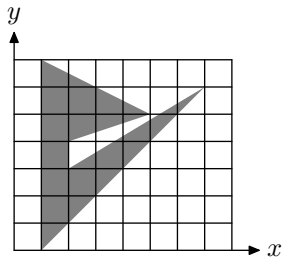
ASCII art is an art of creating pictures with a grid of ASCII characters. There are many styles of ASCII art, but we are interested in the most primitive one, where just an overall character density is used to represent differently shaded areas of the picture.

You should write a proof-of-concept program that renders a filled closed polygon with a rectangular grid of ASCII characters. The whole process is explained in detail below.

Let OXY be a Cartesian coordinate system with OX pointing to the right and OY pointing up. Drawing canvas is bounded with $(0, 0)$ – (w, h) rectangle. Pixels on the canvas are (x, y) – $(x + 1, y + 1)$ squares where x and y are integers such that $0 \le x < w$ and $0 \le y < h$. A filled closed polygon without self-intersections and self-touchings (but not necessarily convex) is drawn on the canvas. Pixels of the canvas become partially filled during the process. Each pixel is represented by an ASCII character depending on the percentage of its filled area according to the following table:

Pixel percentage area filled	*Character name*	*Glyph*	*ASCII code*
From 0% inclusive to 25% exclusive	Full stop	`.`	46
From 25% inclusive to 50% exclusive	Plus sign	`+`	43
From 50% inclusive to 75% exclusive	Small letter o	`o`	111
From 75% inclusive to 100% exclusive	Dollar sign	`$`	36
100%	Number sign	`#`	35

The resulting ASCII characters for all pixels are printed top-to-bottom and left-to-right to get a visual representation of the drawing.



Input

The first line of the input file contains integers n, w, and h ($3 \le n \le 100$, $1 \le w, h \le 100$) — number of vertices in the polygon, width and height of the canvas respectively. The following n lines contain coordinates of the polygon vertices in clockwise order. Point i is described by two integers x_i and y_i ($0 \le x_i \le w$, $0 \le y_i \le h$).

Output

Write to the output file h lines with w ASCII characters each that represent ASCII art drawing of the given polygon.

Sample input and output

ascii.in	ascii.out
6 8 7 7 6 1 0 1 7 5 5 2 4 2 3	.$+..... .##$+... .#$oo+.. .#+$o... .##o.... .#o..... .o.....

Problem B. Billing Tables

Input file: `billing.in`
Output file: `billing.out`

In the world of telecommunications phone calls to different phone numbers have to be charged using different rate or different billing plan. International Carrier of Phone Communications (ICPC) has an antique billing table that determines which phone call has to be charged using which billing plan.

Each international phone number has 11 digits. The billing table has n lines. Each line specifies a range of prefixes of phone numbers like "`7919 - 921`". This specification means that all phone numbers starting from 7919, 7920, and 7921 match this line. A billing plan name is specified for each prefix. To determine a billing plan for a call, the table is scanned from top to bottom and the first matching line determines the billing plan. If no match is found, the phone number is invalid and no billing plan is needed. A special billing plan named "`invalid`" (without quotes) is used as an alternative way to define invalid phone numbers. Some billing plans are used for quite differently looking phone numbers and their names may be specified on different lines in different places of the table.

ICPC's billing table is old and contains many entries. Some of those entries may not be even used anymore. It is very hard to figure out which phone numbers each billing plan is actually used for. The ICPC's management has reached a decision to transform this billing table into a more legible format. In this new format table consists of the lexicographically ordered list of simple prefixes (without the "`-`" range feature of the old format) with a billing plan name for each prefix. No prefix of this new billing table should be a prefix of any other prefix from the table. Thus, a simple dictionary lookup (binary search, for example) will be sufficient to figure out a billing plan for a given phone number. Finding all phone numbers for a given billing plan will also become quite a simple task. The number of lines in the new billing table should be minimized. Billing plan named "`invalid`" should not be present in the new billing table at all, since invalid phone numbers will be denoted by absence of the corresponding prefix in the new billing table.

Input

The first line of the input file contains a single integer number n ($1 \le n \le 100$) — the number of lines in the old billing table. The following n lines describe the old billing table with one rule on a line. Each rule contains four tokens separated by spaces — prefix A, minus sign ("`-`"), prefix B, and billing plan name. Prefixes contain from 1 to 11 digits each, and the billing plan name contains from 1 to 20 lower case letters.

Further, let us denote with $|A|$ the number of digits in the prefix A. It is true that $1 \le |B| \le |A| \le 11$. Moreover, last $|B|$ digits of prefix A form a string that is lexicographically equal or precedes B.

Such pair of prefixes A and B matches all phone numbers with the first $|A| - |B|$ digits matching the first digits of A and with the following $|B|$ digits being lexicographically between the last $|B|$ digits of A and B (inclusive).

Output

Write to the output file a single integer number k — the minimal number of lines that the new table should contain to describe the given old billing table. Then write k lines with the lexicographically ordered new billing table. Write two tokens separated by a space on each line — the prefix and the billing plan name. Note, that the prefix in the new billing table shall contain at least one digit.

If all phone numbers are invalid (every phone number has no matching line or matches line with billing plan "`invalid`") then the output file should contain just number zero.

Sample input and output

billing.in	billing.out
8 7919 - 921 cell 7921800 - 999 priv 1 - 1 usa 760 - 9 rsv 7928 - 29 rsv 7600 - 7899 spec 73 - 77 invalid 7 - 7 cis	35 1 usa 70 cis 71 cis 72 cis 76 rsv 77 spec 78 spec 790 cis 7910 cis 7911 cis 7912 cis 7913 cis 7914 cis 7915 cis 7916 cis 7917 cis 7918 cis 7919 cell 7920 cell 7921 cell 7922 cis 7923 cis 7924 cis 7925 cis 7926 cis 7927 cis 7928 rsv 7929 rsv 793 cis 794 cis 795 cis 796 cis 797 cis 798 cis 799 cis

Problem C. Cellular Automaton

Input file: `cell.in`
Output file: `cell.out`

A *cellular automaton* is a collection of cells on a grid of specified shape that evolves through a number of discrete time steps according to a set of rules that describe the new state of a cell based on the states of neighboring cells. The *order of the cellular automaton* is the number of cells it contains. Cells of the automaton of order n are numbered from 1 to n.

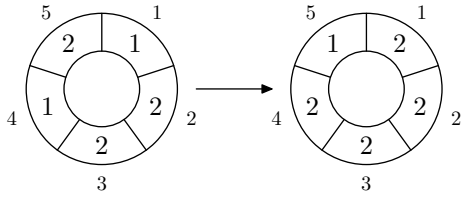
The *order of the cell* is the number of different values it may contain. Usually, values of a cell of order m are considered to be integer numbers from 0 to $m - 1$.

One of the most fundamental properties of a cellular automaton is the type of grid on which it is computed. In this problem we examine the special kind of cellular automaton — circular cellular automaton of order n with cells of order m. We will denote such kind of cellular automaton as *n,m-automaton*.

A distance between cells i and j in n,m-automaton is defined as $\min(|i-j|, n-|i-j|)$. A *d-environment of a cell* is the set of cells at a distance not greater than d.

On each *d-step* values of all cells are simultaneously replaced by new values. The new value of cell i after d-step is computed as a sum of values of cells belonging to the d-enviroment of the cell i modulo m.

The following picture shows 1-step of the 5,3-automaton.



The problem is to calculate the state of the n,m-automaton after k d-steps.

Input

The first line of the input file contains four integer numbers n, m, d, and k ($1 \le n \le 500$, $1 \le m \le 1\,000\,000$, $0 \le d < \frac{n}{2}$, $1 \le k \le 10\,000\,000$). The second line contains n integer numbers from 0 to $m - 1$ — initial values of the automaton's cells.

Output

Output the values of the n,m-automaton's cells after k d-steps.

Sample input and output

cell.in	cell.out
5 3 1 1 1 2 2 1 2	2 2 2 2 1
5 3 1 10 1 2 2 1 2	2 0 0 2 2

Problem D. Driving Directions

Input file: `driving.in`
Output file: `driving.out`

Contrary to the popular belief, alien flying saucers cannot fly arbitrarily around our planet Earth. Their touch down and take off maneuvers are extremely energy consuming, so they carefully plan their mission to Earth to touch down in one particular place, then hover above the ground carrying out their mission, then take off. It was all so easy when human civilization was in its infancy, since flying saucers can hover above all the trees and building, and their shortest path from one mission point to the other was usually a simple straight line — the most efficient way to travel. However, modern cities have so tall skyscrapers that flying saucers cannot hover above them and the task of navigating modern city became quite a complex one. You were hired by an alien spy to write a piece of software that will ultimately give flying saucers driving directions throughout the city. As your first assignment (to prove your worth to your alien masters) you should write a program that computes the shortest distance for a flying saucer from one point to another. This program will be used by aliens as an aid in planning of mission energy requirements.

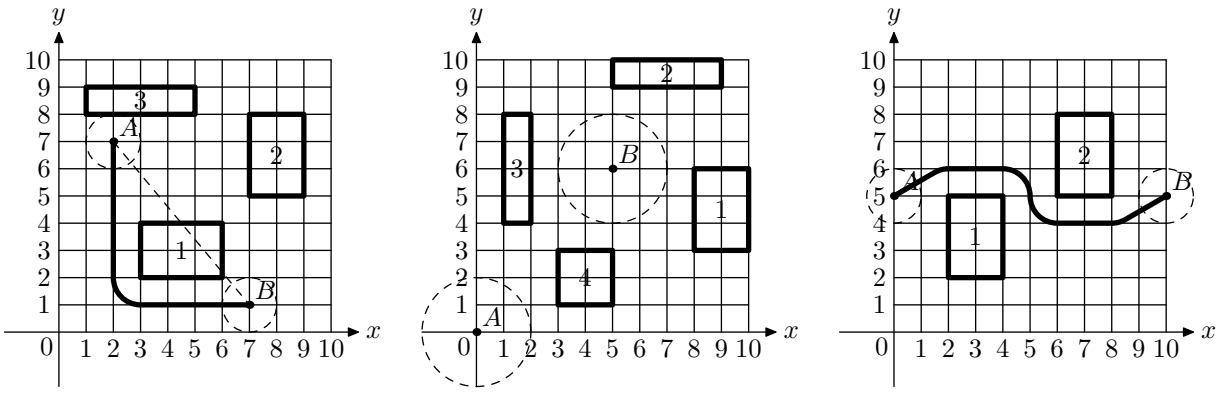
The problem is simplified by several facts. First of all, since flying saucer can hover above most of the buildings, you are only concerned with locations of skyscrapers. Second, the problem is actually two-dimensional — you can look at everything "from above" and pretend that all objects are situated on OXY Cartesian plane. Flying saucer is represented by a circle of radius r, and since modern cities with skyscrapers tend to be regular, every skyscraper is represented with a rectangle whose sides are parallel to OX and OY axes.

By definition, the location of flying saucer is the location of its center, and the length of the path it travels is the length of the path its center travels. During its mission flying saucer can touch skyscrapers but it cannot intersect them.

At the first picture a flying saucer of $r = 1$ has to get from point A to point B. The straight dashed line would have been the shortest path if not for skyscraper 1. The shortest way to avoid skyscraper 1 is going around its top right corner, but skyscraper 2 is too close to fly there. Thus, the answer is to go around the bottom left corner of skyscraper 1 for a total path length of 10.570796.

In the second picture it is impossible for a flying saucer of $r = 2$ to get from point A to point B, since all skyscrapers are too close to fly in between them.

In the third picture flying saucer of $r = 1$ has to fly in a slalom-like way around two skyscrapers in order to achieve the shortest path of length 11.652892 between A and B.



Input

The first line of the input file contains integer numbers r and n ($1 \le r \le 100$, $0 \le n \le 30$), where r is the radius of the flying saucer, and n is the number of skyscrapers. The next line contains four integer

numbers x_A, y_A, x_B, and y_B ($-1000 \le x_A, y_A, x_B, y_B \le 1000$), where (x_A, y_A) are the coordinates of the starting point of the flying saucer's mission and (x_B, y_B) are the coordinates of its finishing point.

The following n lines describe skyscrapers. Each skyscraper is represented by four integer numbers x_1, y_1, x_2, and y_2 ($-1000 \le x_1, y_1, x_2, y_2 \le 1000$, $x_1 < x_2$, $y_1 < y_2$) — coordinates of the corners of the corresponding rectangle.

Skyscrapers neither intersect nor touch each other. Starting and finishing points of the flying saucer's mission are valid locations for flying saucer, that is, it does not intersect any skyscraper in those points, but may touch some of them.

Output

Write to the output file text "`no solution`" (without quotes) if the flying saucer cannot reach its finishing point from the starting one. Otherwise, write to the output file a single number — the shortest distance that the flying saucer needs to travel to get from the starting point to the finishing point. Answer has to be precise to at least 6 digits after the decimal point.

Sample input and output

driving.in	driving.out
1 3 2 7 7 1 3 2 6 4 7 5 9 8 1 8 5 9	10.570796
2 4 0 0 5 6 8 3 10 6 5 9 9 10 1 4 2 8 3 1 5 3	no solution
1 2 0 5 10 5 2 2 4 5 6 5 8 8	11.652892

Problem E. Exchange

Input file: `exchange.in`
Output file: `exchange.out`

You are taking part in a large project to automate operations for Northeastern Exchange of Resources and Commodities (NEERC). Different resources and commodities are traded on this exchange via public auction. Each resource or commodity is traded independently of the others and your task is to write a core engine for this exchange — its order book. There is a separate instance of an order book for each traded resource or commodity and it is not your problem to get the correct orders into order books. The order book instance you will be writing is going to receive the appropriate orders from the rest of exchange system.

Order book receives a stream of *messages*. Messages are *orders* and requests to cancel previously issued orders. Orders that were not cancelled are called *active*. There are orders *to buy* and orders *to sell*. Each order to buy or to sell has a positive *size* and a positive *price*. Order book maintains a list of active orders and generates *quotes* and *trades*. Active order to buy at the highest price is the best buy order and its price is called *bid* price. Active order to sell at the lowest price is the best sell order and its price is called *ask* price. Ask price is always lower than bid price, that is, buyers are willing to pay less than sellers want to receive in return.

A current quote from the order book contains current bid size, bid price, ask size, and ask price. Here bid and ask sizes are sums of the the sizes of all active orders with the current bid price and the current ask price correspondingly.

A trade records information about transaction between buyer and seller. Each trade has size and price.

If an order to buy arrives to the order book at a price greater or equal to the current ask price, then the corresponding orders are *matched* and trade happens — buyer and seller reached agreement on a price. Vice versa, if an order to sell arrives to the order book at a price less or equal to the current bid price, then trade happens, too. For the purpose of order matching, order book works like a FIFO queue for orders with the same price (read further for details).

When an order to buy arrives to the order book at a price greater or equal to the current ask price it is not immediately entered into the order book. First, a number of trades is generated, possibly reducing the size of incoming order. Trade is generated between incoming buy order and the best order to sell. If there are multiple best orders (at the ask price), then the order that entered the order book first is chosen. Trade is generated at the current ask price with the size of the trade being equal to the smaller of the sizes of two matching orders. Sizes of both matching orders are reduced by the size of the trade. If that reduces the size of sell order to zero, then it becomes inactive and is removed from the order book. If the size of incoming buy order becomes zero, then the process is over — incoming order becomes inactive. If the size of incoming buy order is still positive and there is another sell order to match with, then the process continues generating further trades at the new ask price (ask price can increase as sell orders are traded against and become inactive). If there is no sell order to match with (current ask price became greater than incoming buy order price), then incoming buy order is added to the order book with its remaining size.

For incoming sell order everything works similarly – it is matched with buy orders from the order book and trades are generated on bid price.

On incoming cancel request the corresponding order is simply removed from the order book and becomes inactive. Note, that by the time of the cancel request the quantity of the corresponding order might have been already partially reduced or the order might have become inactive. Requests to cancel inactive order do not change anything in the order book.

On every incoming message the order book has to generate all trades it causes and the current quote (bid size, bid price, ask size, ask price) after processing of the corresponding message, even when nothing has changed in the order book as a result of this message. Thus, the number of quotes the order book

generates is always equal to the number of incoming messages.

Input

The first line of the input file contains a single integer number n ($1 \le n \le 10\,000$) — the number of incoming messages that the order book has to process. The following n lines contain messages. Each line starts with a word describing the message type — `BUY`, `SELL`, or `CANCEL` followed after a space by the message parameters.

`BUY` and `SELL` denote an order to buy or to sell correspondingly, and are followed by two integers q and p ($1 \le q \le 99\,999$, $1 \le p \le 99\,999$) — order size and price. `CANCEL` denotes a request to cancel previously issued order. It is followed by a single integer i which is the number of the message with some preceding order to buy or to sell (messages are numbered from 1 to n).

Output

Write to the output file a stream of quotes and trades that the incoming messages generate. For every trade write `TRADE` followed after space by the trade size and price. For every quote write `QUOTE` followed after space by the quote bid size, bid price, minus sign ("-"), ask size, ask price (all separated by spaces).

There is a special case when there are no active orders to buy or to sell in the order book (bid and/or ask are not defined). This case is treated as follows. If there is no active order to buy, then it is assumed that bid size is zero and bid price is zero. If there is no active order to sell, then it is assumed that ask size is zero and ask price is 99 999. Note, that zero is not a legal price, but 99 999 is a legal price. Recipient of quote messages distinguishes actual 99 999 ask price from the special case of absent orders to sell by looking at its ask size.

See example for further clarification.

Sample input and output

exchange.in	exchange.out
11 BUY 100 35 CANCEL 1 BUY 100 34 SELL 150 36 SELL 300 37 SELL 100 36 BUY 100 38 CANCEL 4 CANCEL 7 BUY 200 32 SELL 500 30	QUOTE 100 35 - 0 99999 QUOTE 0 0 - 0 99999 QUOTE 100 34 - 0 99999 QUOTE 100 34 - 150 36 QUOTE 100 34 - 150 36 QUOTE 100 34 - 250 36 TRADE 100 36 QUOTE 100 34 - 150 36 QUOTE 100 34 - 100 36 QUOTE 100 34 - 100 36 QUOTE 100 34 - 100 36 TRADE 100 34 TRADE 200 32 QUOTE 0 0 - 200 30

Problem F. Fool's Game

Input file: `fool.in`
Output file: `fool.out`

A card game, often called "Fool's Game", is quite popular in Russia. We will describe a game for two players. A standard deck of 36 cards is used. One suit is declared to be a *trump*.

A game consists of rounds. Before the round each player has several cards, one of the players is *starting*, the other one is *covering*. The starting player starts by laying one or several cards of the same rank down on the table. The number of cards must not exceed the number of cards the covering player has. The covering player must in turn *cover* all the cards with some of her cards, laying them on the table above the uncovered cards. A card can cover another if one of the following is true:

- it has the same suit and higher rank (ranks are ordered as usually: 6, 7, 8, 9, 10, J, Q, K, A);
- it is a trump and the card to cover is not a trump (a trump can only be covered by a higher trump).

After the cards on the table are all covered, the starting player can *toss* some more cards to be covered. The rank of each card tossed must be among the ranks of the cards already on the table at the moment. Now the newly added cards must be covered by the covering player, after that the starting player can toss more cards, and so on. The starting player cannot toss more cards than the covering player has at the moment.

The round ends when either the covering player cannot or does not want to cover all uncovered cards on the table, or when the starting player cannot or does not want to toss more cards.

In the first case, when the covering player declares that she does not want to cover all uncovered cards on the table, the starting player is given a chance to toss in more cards. The ranks of the cards tossed must be among the ranks of the cards already on the table. The number of uncovered cards on the table cannot exceed the number of cards that the covering player has at the moment. After that, the covering player loses the round and takes all the cards from the table, adding them to her cards. Starting player keeps her starting role and moves again in the next round.

In the second case, when all cards on the table are covered and the starting player cannot or does not want to toss more cards, the covering player wins the round and the cards on the table are removed from the game. The players' roles for the next round are swapped: the covering player becomes the starting one and vice versa.

If, after the end of the round, one of the players has no cards, and the other one has one or more cards, then the player with no cards wins the game. If both players have no cards, then the player who was starting in the last round wins the game.

Given the trump suit and the cards the players initially have, find out who wins the game if both play optimally. Both players have full information about cards in the game.

Input

The first line of the input file contains n_1 and n_2 — the number of cards that each of the players has in the beginning of the round ($1 \le n_1, n_2 \le 6$), and the trump suit (suit is specified using one letter: '`S`' for spades, '`C`' for clubs, '`D`' for diamonds, '`H`' for hearts).

The second line contains n_1 card descriptions — the cards of the first player. Each card is specified by its rank ('`6`'...'`9`', '`T`' for 10, '`J`' for Jack, '`Q`' for Queen, '`K`' for King, '`A`' for Ace) followed by its suit. The third line contains n_2 card descriptions — the cards of the covering player. The first player is the starting player in the first round.

All cards in players' hands are different.

Output

Output "`FIRST`" if the first player wins the game, or "`SECOND`" if the second player does.

Sample input and output

fool.in	fool.out
2 2 S KC AD 6S 7S	SECOND
2 2 D KC AD 6S 7S	FIRST
4 5 C AS 6S 7S 8S 9S TS JS QS KS	SECOND
3 2 C 6H JS JD AD 6C	FIRST

Problem G. Graveyard

Input file: `graveyard.in`
Output file: `graveyard.out`

Programming contests became so popular in the year 2397 that the governor of New Earck — the largest human-inhabited planet of the galaxy — opened a special Alley of Contestant Memories (ACM) at the local graveyard. The ACM encircles a green park, and holds the holographic statues of famous contestants placed equidistantly along the park perimeter. The alley has to be renewed from time to time when a new group of memorials arrives.

When new memorials are added, the exact place for each can be selected arbitrarily along the ACM, but the equidistant disposition must be maintained by moving some of the old statues along the alley.

Surprisingly, humans are still quite superstitious in 24th century: the graveyard keepers believe the holograms are holding dead people souls, and thus always try to renew the ACM with minimal possible movements of existing statues (besides, the holographic equipment is very heavy). Statues are moved along the park perimeter. Your work is to find a renewal plan which minimizes the sum of travel distances of all statues. Installation of a new hologram adds no distance penalty, so choose the places for newcomers wisely!

Input

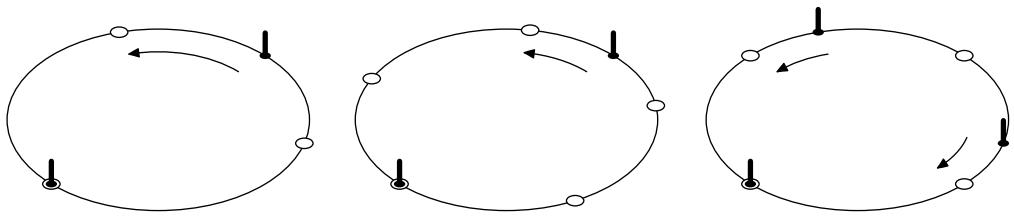
Input file contains two integer numbers: n — the number of holographic statues initially located at the ACM, and m – the number of statues to be added ($2 \le n \le 1000, 1 \le m \le 1000$). The length of the alley along the park perimeter is exactly 10 000 feet.

Output

Write a single real number to the output file — the minimal sum of travel distances of all statues (in feet). The answer must be precise to at least 4 digits after decimal point.

Sample input and output

graveyard.in	graveyard.out
2 1	1666.6667
2 3	1000.0
3 1	1666.6667
10 10	0.0



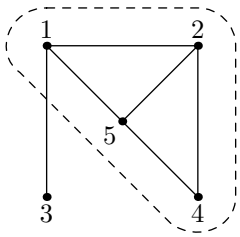
Pictures show the first three examples. Marked circles denote original statues, empty circles denote new equidistant places, arrows denote movement plans for existing statues.

Problem H. Hard Life

Input file: `hard.in`
Output file: `hard.out`

John is a Chief Executive Officer at a privately owned medium size company. The owner of the company has decided to make his son Scott a manager in the company. John fears that the owner will ultimately give CEO position to Scott if he does well on his new manager position, so he decided to make Scott's life as hard as possible by carefully selecting the team he is going to manage in the company.

John knows which pairs of his people work poorly in the same team. John introduced a *hardness factor* of a team — it is a number of pairs of people from this team who work poorly in the same team divided by the total number of people in the team. The larger is the hardness factor, the harder is this team to manage. John wants to find a group of people in the company that are harderst to manage and make it Scott's team. Please, help him.



In the example on the picture the hardest team consists of people 1, 2, 4, and 5. Among 4 of them 5 pairs work poorly in the same team, thus hardness factor is equal to $\frac{5}{4}$. If we add person number 3 to the team then hardness factor decreases to $\frac{6}{5}$.

Input

The first line of the input file contains two integer numbers n and m ($1 \le n \le 100$, $0 \le m \le 1000$). Here n is a total number of people in the company (people are numbered from 1 to n), and m is the number of pairs of people who work poorly in the same team. Next m lines describe those pairs with two integer numbers a_i and b_i ($1 \le a_i, b_i \le n$, $a_i \ne b_i$) on a line. The order of people in a pair is arbitrary and no pair is listed twice.

Output

Write to the output file an integer number k ($1 \le k \le n$) — the number of people in the hardest team, followed by k lines listing people from this team in ascending order. If there are multiple teams with the same hardness factor then write any one.

Sample input and output

hard.in	hard.out
5 6 1 5 5 4 4 2 2 5 1 2 3 1	4 1 2 4 5
4 0	1 1

Note, that in the last example any team has hardness factor of zero, and any non-empty list of people is a valid answer.

Problem I. Interconnect

Input file: `interconnect.in`
Output file: `interconnect.out`

There are two serious problems in the Kingdom of Lipshire: the roads and the fools who build them. Once upon a time, the King of Lipshire has decided to improve the road system because some roads became completely impassable — it was easier to travel cross-country instead of using those roads.

By King's decree, new roads are to be built in Lipshire. Of course, the new road system must interconnect all towns, i. e. there must be a path connecting any two towns of Lipshire.

The road administration of Lipshire has resources to build exactly one road per year. Unfortunately, the fools who build these roads are completely out of control. So, regardless of the orders given, the fools randomly select two different towns a and b and build a road between them, even when those towns are already connected by a road. All possible choices are equiprobable. The road is build in such a manner that the only points where a traveler can leave it are the towns connected by this road. The only good thing is that all roads are bidirectional.

The King knows about the problem, but he cannot do anything about it. The only thing King needs to know is the expected number of years to wait before the road system of Lipshire becomes interconnected. He asked you to provide this information.

Input

The first line of the input contains two integers n and m ($2 \le n \le 30$, $0 \le m \le 1\,000$) — the number of towns in Lipshire, and the number of roads which are still good. The following m lines describe roads, one per line. Each road is described with two endpoints — two integer numbers u_i and v_i ($1 \le u_i, v_i \le n$, $u_i \ne v_i$). There can be multiple roads between two towns, but the road from a town to itself is not allowed.

Output

Output the expected number of years to wait for the interconnected road system. If the system is already interconnected, output zero as an answer. Output the number with at least six precise digits after the decimal point.

Sample input and output

interconnect.in	interconnect.out
2 1 1 2	0.0
4 2 1 2 3 4	1.5

Problem J. Java vs C++

Input file: `java_c.in`
Output file: `java_c.out`

Apologists of Java and C++ can argue for hours proving each other that their programming language is the best one. Java people will tell that their programs are clearer and less prone to errors, while C++ people will laugh at their inability to instantiate an array of generics or tell them that their programs are slow and have long source code.

Another issue that Java and C++ people could never agree on is identifier naming. In Java a multiword identifier is constructed in the following manner: the first word is written starting from the small letter, and the following ones are written starting from the capital letter, no separators are used. All other letters are small. Examples of a Java identifier are `javaIdentifier`, `longAndMnemonicIdentifier`, `name`, `nEERC`.

Unlike them, C++ people use only small letters in their identifiers. To separate words they use underscore character '_'. Examples of C++ identifiers are `c_identifier`, `long_and_mnemonic_identifier`, `name` (you see that when there is just one word Java and C++ people agree), `n_e_e_r_c`.

You are writing a translator that is intended to translate C++ programs to Java and vice versa. Of course, identifiers in the translated program must be formatted due to its language rules — otherwise people will never like your translator.

The first thing you would like to write is an identifier translation routine. Given an identifier, it would detect whether it is Java identifier or C++ identifier and translate it to another dialect. If it is neither, then your routine should report an error. Translation must preserve the order of words and must only change the case of letters and/or add/remove underscores.

Input

The input file consists of one line that contains an identifier. It consists of letters of the English alphabet and underscores. Its length does not exceed 100.

Output

If the input identifier is Java identifier, output its C++ version. If it is C++ identifier, output its Java version. If it is none, output "`Error!`" instead.

Sample input and output

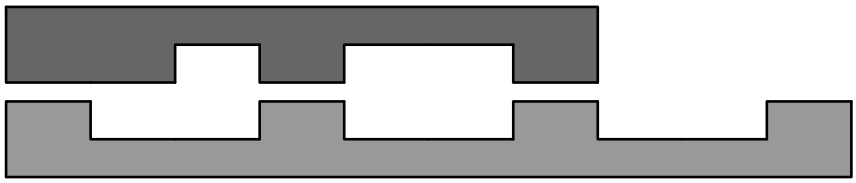
java_c.in	java_c.out
long_and_mnemonic_identifier	longAndMnemonicIdentifier
anotherExample	another_example
i	i
bad_Style	Error!

Problem K. Kickdown

Input file: `kickdown.in`
Output file: `kickdown.out`

A research laboratory of a world-leading automobile company has received an order to create a special transmission mechanism, which allows for incredibly efficient kickdown — an operation of switching to lower gear. After several months of research engineers found that the most efficient solution requires special gears with teeth and cavities placed non-uniformly. They calculated the optimal flanks of the gears. Now they want to perform some experiments to prove their findings.

The first phase of the experiment is done with planar toothed sections, not round-shaped gears. A section of length n consists of n units. The unit is either a cavity of height h or a tooth of height $2h$. Two sections are required for the experiment: one to emulate master gear (with teeth at the bottom) and one for the driven gear (with teeth at the top).



There is a long stripe of width $3h$ in the laboratory and its length is enough for cutting two engaged sections together. The sections are irregular but they may still be put together if shifted along each other.

The stripe is made of an expensive alloy, so the engineers want to use as little of it as possible. You need to find the minimal length of the stripe which is enough for cutting both sections simultaneously.

Input

There are two lines in the input file, each contains a string to describe a section. The first line describes master section (teeth at the bottom) and the second line describes driven section (teeth at the top). Each character in a string represents one section unit — 1 for a cavity and 2 for a tooth. The sections can not be flipped or rotated.

Each string is non-empty and its length does not exceed 100.

Output

Write a single integer number to the output file — the minimal length of the stripe required to cut off given sections.

Sample input and output

kickdown.in	kickdown.out
2112112112 2212112	10
12121212 21212121	8
2211221122 21212	15