Problem A. Ancient Cipher

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

Ancient Roman empire had a strong government system with various departments, including a secret service department. Important documents were sent between provinces and the capital in encrypted form to prevent eavesdropping. The most popular ciphers in those times were so called *substitution cipher* and *permutation cipher*.

Substitution cipher changes all occurrences of each letter to some other letter. Substitutes for all letters must be different. For some letters substitute letter may coincide with the original letter. For example, applying substitution cipher that changes all letters from '`A`' to '`Y`' to the next ones in the alphabet, and changes '`Z`' to '`A`', to the message "`VICTORIOUS`" one gets the message "`WJDUPSJPVT`".

Permutation cipher applies some permutation to the letters of the message. For example, applying the permutation $\langle 2, 1, 5, 4, 3, 7, 6, 10, 9, 8\rangle$ to the message "`VICTORIOUS`" one gets the message "`IVOTCIRSUO`".

It was quickly noticed that being applied separately, both substitution cipher and permutation cipher were rather weak. But when being combined, they were strong enough for those times. Thus, the most important messages were first encrypted using substitution cipher, and then the result was encrypted using permutation cipher. Encrypting the message "`VICTORIOUS`" with the combination of the ciphers described above one gets the message "`JWPUDJSTVP`".

Archeologists have recently found the message engraved on a stone plate. At the first glance it seemed completely meaningless, so it was suggested that the message was encrypted with some substitution and permutation ciphers. They have conjectured the possible text of the original message that was encrypted, and now they want to check their conjecture. They need a computer program to do it, so you have to write one.

Input

Input file contains two lines. The first line contains the message engraved on the plate. Before encrypting, all spaces and punctuation marks were removed, so the encrypted message contains only capital letters of the English alphabet. The second line contains the original message that is conjectured to be encrypted in the message on the first line. It also contains only capital letters of the English alphabet.

The lengths of both lines of the input file are equal and do not exceed 100.

Output

Output "`YES`" if the message on the first line of the input file could be the result of encrypting the message on the second line, or "`NO`" in the other case.

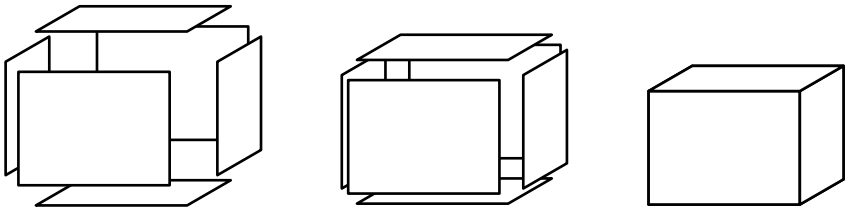
Sample input and output

ancient.in	ancient.out
JWPUDJSTVP VICTORIOUS	YES
MAMA ROME	NO
HAHA HEHE	YES
AAA AAA	YES
NEERCISTHEBEST SECRETMESSAGES	NO

Problem B. Box

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

Ivan works at a factory that produces heavy machinery. He has a simple job — he knocks up wooden boxes of different sizes to pack machinery for delivery to the customers. Each box is a rectangular parallelepiped. Ivan uses six rectangular wooden pallets to make a box. Each pallet is used for one side of the box.



Joe delivers pallets for Ivan. Joe is not very smart and often makes mistakes — he brings Ivan pallets that do not fit together to make a box. But Joe does not trust Ivan. It always takes a lot of time to explain Joe that he has made a mistake.

Fortunately, Joe adores everything related to computers and sincerely believes that computers never make mistakes. Ivan has decided to use this for his own advantage. Ivan asks you to write a program that given sizes of six rectangular pallets tells whether it is possible to make a box out of them.

Input

Input file consists of six lines. Each line describes one pallet and contains two integer numbers w and h ($1 \le w, h \le 10\,000$) — width and height of the pallet in millimeters respectively.

Output

Write a single word "`POSSIBLE`" to the output file if it is possible to make a box using six given pallets for its sides. Write a single word "`IMPOSSIBLE`" if it is not possible to do so.

Sample input and output

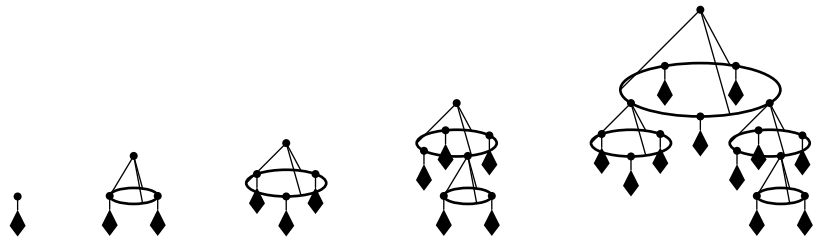
box.in	box.out
1345 2584 2584 683 2584 1345 683 1345 683 1345 2584 683	POSSIBLE
1234 4567 1234 4567 4567 4321 4322 4567 4321 1234 4321 1234	IMPOSSIBLE

Problem C. Chandelier

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

Lamps-O-Matic company assembles very large chandeliers. A chandelier consists of multiple levels. On the first level crystal pendants are attached to the rings. Assembled rings and new pendants are attached to the rings of the next level, and so on. At the end there is a single large ring — the complete chandelier with multiple smaller rings and pendants hanging from it.

A special-purpose robot assembles chandeliers. It has a supply of crystal pendants and empty rings, and a stack to store elements of a chandelier during assembly. Initially the stack is empty. Robot executes a list of commands to assemble a chandelier.



On command "`a`" robot takes a new crystal pendant and places it on the top of the stack. On command "`1`" to "`9`" robot takes the corresponding number of items from the top of the stack and consecutively attaches them to the new ring. The newly assembled ring is then placed on the top of the stack. At the end of the program there is a single item on the stack — the complete chandelier.

Unfortunately, for some programs it turns out that the stack during their execution needs to store too many items at some moments. Your task is to optimize the given program, so that the overall design of the respective chandelier remains the same, but the maximal number of items on the stack during the execution is minimal. A pendant or any complex multi-level assembled ring count as a single item of the stack.

The design of a chandelier is considered to be the same if each ring contains the same items in the same order. Since rings are circular it does not matter what item is on the top of the stack when the robot receives a command to assemble a new ring, but the relative order of the items on the stack is important. For example, if the robot receives command "`4`" when items $\langle i_1, i_2, i_3, i_4 \rangle$ are on the top of the stack in this order (i_1 being the topmost), then the same ring is also assembled if these items are arranged on the stack in the following ways: $\langle i_2, i_3, i_4, i_1 \rangle$, or $\langle i_3, i_4, i_1, i_2 \rangle$, or $\langle i_4, i_1, i_2, i_3 \rangle$.

Input

The input file contains a single line with a valid program for the robot. The program consists of at most 10 000 characters.

Output

On the first line of the output file write the minimal required stack capacity (number of items it can hold) to assemble the chandelier. On the second line write some program for the assembly robot that uses stack of this capacity and results in the same chandelier.

Sample input and output

chandelier.in	chandelier.out
aaaaa3aaa2aaa45	6 aaa3aaa2aaa4aa5

Problem D. Document Indexing

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

Andy is fond of old computers. He loves everything about them and he uses emulators of old operating systems on his modern computer. Andy also likes writing programs for them. Recently he has decided to write a text editor for his favorite text-mode operating system.

The most difficult task he has got stuck with is document indexing. An *index* of the document is the lexicographically ordered list of all words occurring in the document with the numbers of pages they occur at. Andy feels that he is not able to write the component of the editor that performs indexing, so he asks you to help.

A document is a sequence of paragraphs. Each paragraph consists of one or more lines. Paragraphs are separated from each other with exactly one blank line.

First, the document is *paginated* — divided into pages. Each page consists of up to n lines. Lines are placed on the page one after another, until n lines are placed. The following correction rules are then applied:

- If the last line on a page is the last line of the paragraph, then the following empty line is skipped, i.e. it is not placed on any page. Therefore, the page never starts with a blank line.
- If the last line on a page is the first line of a paragraph that contains more than one line (so called *orphan line*), then it is moved to the next page.
- If the last line on a page is the next-to-last line of a paragraph that contains more than three lines, then this line is moved to the next page (otherwise, the last line of the paragraph would be alone on the page — so called *widow line*).
- If the last line on a page is the next-to-last line of a paragraph that contains exactly two or three lines, then the whole paragraph is moved to the next page (so we have neither orphan, nor widow lines).

After applying the correction rules the next page is formed, and so on until the whole document is paginated.

A word is a continuous sequence of letters of the English alphabet. Case is not important.

The index of the document contains each word from the document and the list of the pages it occurs at. The numbers of pages a word occurs at must be listed in the ascending order. Numbers must be separated by commas. If a word occurs on three or more consecutive pages, only the first and the last page numbers of this range must be listed, separated by a dash, for example "`3-5,7-10,12,13,15`".

Input

The first line of the input file contains n ($4 \le n \le 100$). The rest of the input file contains the document to be indexed. The size of the input file does not exceed 20 000 bytes.

The line is considered blank if it is completely empty. No line contains leading or trailing spaces. The document does not contain two consecutive blank lines. The first line of the document is not blank. The length of each line of the document does not exceed 200 characters.

Output

Print all words that occur in the given document. Words must be printed in the lexicographical order, one word on a line. After each word print one space followed by the list of pages it occurs at, formatted as described in problem statement. Use capital letters in output.

Sample input and output

document.in	document.out
6 From thousands of teams competing in regional contests held from September to December 2004 world-wide, seventy-five teams will advance to the World Finals in Shanghai, April 3-7, 2005. Awards, prizes, scholarships, and bragging rights will be at stake for some of the world's finest university students of the computing science. Join us for the challenge, camaraderie, and the fun! Become the best of the best of the best in ACM ICPC! ACM ICPC is the best contest!	ACM 3 ADVANCE 1 AND 2,3 APRIL 1 AT 2 AWARDS 2 BE 2 BECOME 3 BEST 3 BRAGGING 2 CAMARADERIE 3 CHALLENGE 3 COMPETING 1 COMPUTING 2 CONTEST 3 CONTESTS 1 DECEMBER 1 FINALS 1 FINEST 2 FIVE 1 FOR 2,3 FROM 1 FUN 3 HELD 1 ICPC 3 IN 1,3 IS 3 JOIN 3 OF 1-3 PRIZES 2 REGIONAL 1 RIGHTS 2 S 2 SCHOLARSHIPS 2 SCIENCE 2 SEPTEMBER 1 SEVENTY 1 SHANGHAI 1 SOME 2 STAKE 2 STUDENTS 2 TEAMS 1 THE 1-3 THOUSANDS 1 TO 1 UNIVERSITY 2 US 3 WIDE 1 WILL 1,2 WORLD 1,2

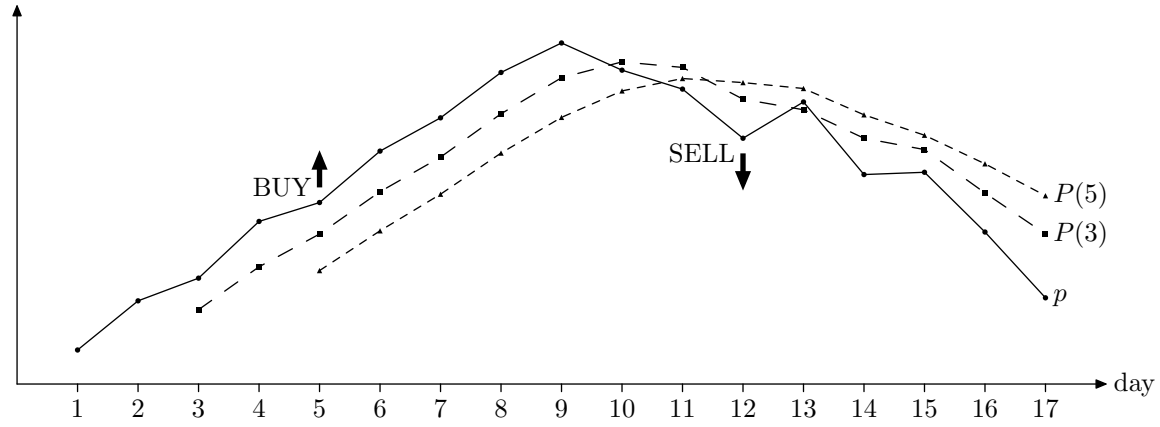
Problem E. Easy Trading

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

Frank is a professional stock trader for *Advanced Commercial Markets Limited* (*ACM Ltd*). He likes "easy trading" — using a straightforward strategy to decide when to buy stock and when to sell it.

Frank has a database of historical stock prices for each day. He uses two integer numbers m and n ($1 \le m < n \le 100$) as parameters of his trading strategy. Every day he computes two numbers: $P(m)$ — an average stock price for the previous m days, and $P(n)$ — an average stock price for the previous n days. $P(m) > P(n)$ is an indicator of the upward trend (traders call it *bullish* trend), and $P(m) < P(n)$ is an indicator of the downward trend (traders call it *bearish* trend). In practice the values for $P(m)$ and $P(n)$ are never equal.

When a trend reverses from bearish to bullish it is a signal for Frank to buy stock. When a trend reverses from bullish to bearish it is a signal to sell.



Frank has different values for m and n in mind and he wants to *backtest* them using historical prices. He takes a set of k ($n < k \le 10\,000$) historical prices p_i ($0 < p_i < 100$ for $1 \le i \le k$). For each i ($n \le i \le k$) he computes $P_i(m)$ and $P_i(n)$ — an arithmetic average of $p_{i-m+1} \ldots p_i$ and $p_{i-n+1} \ldots p_i$ respectively.

Backtesting generates trading signals according to the following rules.

- If $P_i(m) > P_i(n)$ there is a bullish trend for day i and a "`BUY ON DAY` i" signal is generated if $i = n$ or there was a bearish trend on day $i - 1$.
- If $P_i(m) < P_i(n)$ there is a bearish tread for day i and a "`SELL ON DAY` i" signal is generated if $i = n$ or there was a bullish trend on day $i - 1$.

Your task is to write a program that backtests a specified strategy for Frank — you shall print a signal for the first tested day (day n) followed by the signals in increasing day numbers.

Input

The first line of the input file contains three integer numbers m, n, and k. It is followed by k lines with stock prices for days 1 to k. Each stock price p_i is specified with two digits after decimal point. Prices in the input file are such that $P_i(m) \ne P_i(n)$ for all i ($n \le i \le k$).

Output

Write to the output file a list of signals — one signal on a line, as described in the problem statement.

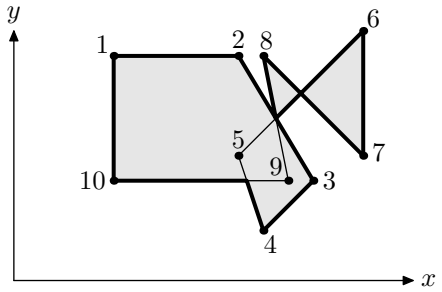
Sample input and output

easy.in	easy.out
3 5 17 8.45 9.10 9.40 10.15 10.40 11.08 11.52 12.12 12.51 12.15 11.90 11.25 11.73 10.77 10.80 10.01 9.14	BUY ON DAY 5 SELL ON DAY 12

Problem F. Find the Border

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

Closed polyline (with possible self-intersections) partitions a plane into a number of regions. One of the regions is unbounded — it is an *exterior* of the polyline. All the bounded regions together with the polyline itself form an *interior* of the polyline (shaded in the picture below). The *border* of the interior (bold line in the picture) is a polyline as well. This polyline has the same interior as the original one. Your task is to find the border of the interior of the given polyline.



To guarantee the uniqueness (up to the starting point) of the polyline representing the border we require that the following conditions are satisfied for it:

- it has no self-intersections, although may have self-touchings;
- no adjacent vertices of the border coincide;
- no adjacent edges of the border are collinear;
- when traversing the border, its interior is always to the left of its edges.

Input

The first line of the input file contains an integer number n ($3 \le n \le 100$) — the number of vertices in the original polyline. Following n lines contain two integer numbers x_i and y_i on a line ($0 \le x_i, y_i \le 100$) — coordinates of the vertices. All vertices are different and no vertex lies on an edge between two other vertices. Adjacent edges of the polyline are not collinear.

Output

Write to the output file an integer number m — the number of vertices of the border. Then write m lines with coordinates of the vertices. Coordinates must be precise up to 4 digits after the decimal point.

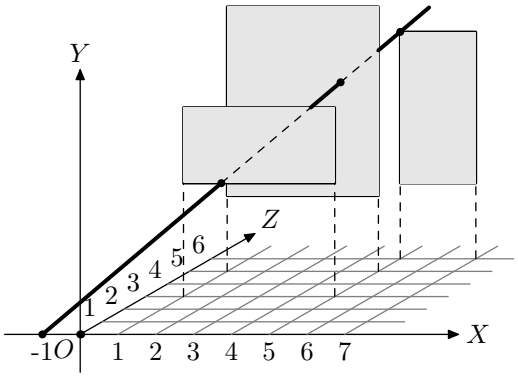
Sample input and output

find.in	find.out
10 4 9 9 9 12 4 10 2 9 5 14 10 14 5 10 9 11 4 4 4	13 9.3333 4 10 2 12 4 10.5 6.5 11.5 7.5 14 5 14 10 11.5 7.5 10 9 10.5 6.5 9 9 4 9 4 4

Problem G. Gunman

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

Consider a 3D scene with $OXYZ$ coordinate system. Axis OX points to the right, axis OY points up, and axis OZ points away from you. There is a number of rectangular windows on the scene. The plane of each window is parallel to OXY, its sides are parallel to OX and OY. All windows are situated at different depths on the scene (different coordinates $z > 0$).



A gunman with a rifle moves along OX axis ($y = 0$ and $z = 0$). He can shoot a bullet in a straight line. His goal is to shoot a single bullet through all the windows. Just touching a window edge is enough.

Your task is to determine how to make such shot.

Input

The first line of the input file contains a single integer number n ($2 \le n \le 100$) — the number of windows on the scene. The following n lines describe the windows. Each line contains five integer numbers x_{1i}, y_{1i}, x_{2i}, y_{2i}, z_i ($0 < x_{1i}, y_{1i}, x_{2i}, y_{2i}, z_i < 1000$). Here (x_{1i}, y_{1i}, z_i) are coordinates of the bottom left corner of the window, and (x_{2i}, y_{2i}, z_i) are coordinates of the top right corner of the window ($x_{1i} < x_{2i}$, $y_{1i} < y_{2i}$). Windows are ordered by z coordinate ($z_i > z_{i-1}$ for $2 \le i \le n$).

Output

Output a single word "`UNSOLVABLE`" if the gunman cannot reach the goal of shooting a bullet through all the windows.

Otherwise, on the first line output a word "`SOLUTION`". On the next line output x coordinate of the point from which the gunman must fire a bullet. On the following n lines output x, y, z coordinates of the points where the bullet goes through the consecutive windows. All coordinates in the output file must be printed with six digits after decimal point.

Sample input and output

gunman.in	gunman.out
3 1 3 5 5 3 1 2 5 7 5 5 2 7 6 6	SOLUTION -1.000000 2.000000 3.000000 3.000000 4.000000 5.000000 5.000000 5.000000 6.000000 6.000000
3 2 1 5 4 1 3 5 6 8 2 4 3 8 6 4	UNSOLVABLE

Problem H. Heapsort

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

A well known algorithm called *heapsort* is a deterministic sorting algorithm taking $O(n \log n)$ time and $O(1)$ additional memory. Let us describe ascending sorting of an array of different integer numbers.

The algorithm consists of two phases. In the first phase, called *heapification*, the array of integers to be sorted is converted to a *heap*. An array $a[1 \dots n]$ of integers is called a heap if for all $1 \le i \le n$ the following *heap conditions* are satisfied:

- if $2i \le n$ then $a[i] > a[2i]$;
- if $2i + 1 \le n$ then $a[i] > a[2i + 1]$.

We can interpret an array as a binary tree, considering children of element $a[i]$ to be $a[2i]$ and $a[2i + 1]$. In this case the parent of $a[i]$ is $a[i \text{ div } 2]$, where $i \text{ div } 2 = \lfloor i/2 \rfloor$. In terms of trees the property of being a heap means that for each node its value is greater than the values of its children.

In the second phase the heap is turned into a sorted array. Because of the heap condition the greatest element in the heapified array is $a[1]$. Let us exchange it with $a[n]$, now the greatest element of the array is at its correct position in the sorted array. This is called *extract-max*.

Now let us consider the part of the array $a[1 \dots n - 1]$. It may be not a heap because the heap condition may fail for $i = 1$. If it is so (that is, either $a[2]$ or $a[3]$, or both are greater than $a[1]$) let us exchange the greatest child of $a[1]$ with it, restoring the heap condition for $i = 1$. Now it is possible that the heap condition fails for the position that now contains the former value of $a[1]$. Apply the same procedure to it, exchanging it with its greatest child. Proceeding so we convert the whole array $a[1 \dots n - 1]$ to a heap. This procedure is called *sifting down*. After converting the part $a[1 \dots n - 1]$ to a heap by sifting, we apply extract-max again, putting second greatest element of the array to $a[n - 1]$, and so on.

For example, let us see how the heap $a = (5, 4, 2, 1, 3)$ is converted to a sorted array. Let us make the first extract-max. After that the array turns to $(3, 4, 2, 1, \mathbf{5})$. Heap condition fails for $a[1] = 3$ because its child $a[2] = 4$ is greater than it. Let us sift it down, exchanging $a[1]$ and $a[2]$. Now the array is $(4, 3, 2, 1, \mathbf{5})$. The heap condition is satisfied for all elements, so sifting is over. Let us make extract-max again. Now the array turns to $(1, 3, 2, \mathbf{4}, \mathbf{5})$. Again the heap condition fails for $a[1]$; exchanging it with its greatest child we get the array $(3, 1, 2, \mathbf{4}, \mathbf{5})$ which is the correct heap. So we make extract-max and get $(2, 1, \mathbf{3}, \mathbf{4}, \mathbf{5})$. This time the heap condition is satisfied for all elements, so we make extract-max, getting $(1, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5})$. The leading part of the array is a heap, and the last extract-max finally gives $(\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5})$.

It is known that heapification can be done in $O(n)$ time. Therefore, the most time consuming operation in heapsort algorithm is sifting, which takes $O(n \log n)$ time.

In this problem you have to find a heapified array containing different numbers from 1 to n, such that when converting it to a sorted array, the total number of exchanges in all sifting operations is maximal possible. In the example above the number of exchanges is $1 + 1 + 0 + 0 + 0 = 2$, which is not the maximum. $(5, 4, 3, 2, 1)$ gives the maximal number of 4 exchanges for $n = 5$.

Input

Input file contains n ($1 \le n \le 50\,000$).

Output

Output the array containing n different integer numbers from 1 to n, such that it is a heap, and when converting it to a sorted array, the total number of exchanges in sifting operations is maximal possible. Separate numbers by spaces.

Sample input and output

heapsort.in	heapsort.out
6	6 5 3 2 4 1

Problem I. Irrelevant Elements

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

Young cryptoanalyst Georgie is investigating different schemes of generating random integer numbers ranging from 0 to $m - 1$. He thinks that standard random number generators are not good enough, so he has invented his own scheme that is intended to bring more randomness into the generated numbers.

First, Georgie chooses n and generates n random integer numbers ranging from 0 to $m - 1$. Let the numbers generated be $a_1, a_2, \ldots, a_n$. After that Georgie calculates the sums of all pairs of adjacent numbers, and replaces the initial array with the array of sums, thus getting $n - 1$ numbers: $a_1 + a_2, a_2 + a_3, \ldots, a_{n-1} + a_n$. Then he applies the same procedure to the new array, getting $n - 2$ numbers. The procedure is repeated until only one number is left. This number is then taken modulo m. That gives the result of the generating procedure.

Georgie has proudly presented this scheme to his computer science teacher, but was pointed out that the scheme has many drawbacks. One important drawback is the fact that the result of the procedure sometimes does not even depend on some of the initially generated numbers. For example, if $n = 3$ and $m = 2$, then the result does not depend on a_2.

Now Georgie wants to investigate this phenomenon. He calls the i-th element of the initial array *irrelevant* if the result of the generating procedure does not depend on a_i. He considers various n and m and wonders which elements are irrelevant for these parameters. Help him to find it out.

Input

Input file contains n and m ($1 \le n \le 100\,000$, $2 \le m \le 10^9$).

Output

On the first line of the output file print the number of irrelevant elements of the initial array for given n and m. On the second line print all such i that i-th element is irrelevant. Numbers on the second line must be printed in the ascending order and must be separated by spaces.

Sample input and output

irrelevant.in	irrelevant.out
3 2	1 2

Problem J. Joke with Turtles

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

There is a famous joke-riddle for children:

> Three turtles are crawling along a road. One turtle says: "There are two turtles ahead of me." The other turtle says: "There are two turtles behind me." The third turtle says: "There are two turtles ahead of me and two turtles behind me." How could this have happened?
>
> The answer is — the third turtle is lying!

Now in this problem you have n turtles crawling along a road. Some of them are crawling in a group, so that they do not see members of their group neither ahead nor behind them. Each turtle makes a statement of the form: "There are a_i turtles crawling ahead of me and b_i turtles crawling behind me." Your task is to find the minimal number of turtles that must be lying.

Let us formalize this task. Turtle i has x_i coordinate. Some turtles may have the same coordinate. Turtle i tells the truth if and only if a_i is the number of turtles such that $x_j > x_i$ and b_i is the number of turtles such that $x_j < x_i$. Otherwise, turtle i is lying.

Input

The first line of the input file contains integer number n ($1 \le n \le 1000$). It is followed by n lines containing numbers a_i and b_i ($0 \le a_i, b_i \le 1000$) that describe statements of each turtle for i from 1 to n.

Output

On the first line of the output file write an integer number m — the minimal number of turtles that must be lying, followed by m integers — turtles that are lying. Turtles can be printed in any order. If there are different sets of m lying turtles, then print any of them.

Sample input and output

joke.in	joke.out
3 2 0 0 2 2 2	1 3
5 0 2 0 3 2 1 1 2 4 0	2 1 4

Problem K. Kingdom of Magic

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

Kingdom of Magic has a network of bidirectional magic portals between cities since ancient times. Each portal magically connects a pair of cities and allows fast magical communication and travel between them. Cities that are connected by the magic portal are called *neighboring*.

Prince Albert and Princess Betty are living in the neighboring cities. Since their childhood Albert and Betty were always in touch with each other using magic communication Orbs, which work via a magic portal between the cities.

Albert and Betty are in love with each other. Their love is so great that they cannot live a minute without each other. They always carry the Orbs with them, so that they can talk to each other at any time. There is something strange about their love — they have never seen each other and they even fear to be in the same city at the same time. People say that the magic of the Orbs have affected them.

Traveling through the Kingdom is a complicated affair for Albert and Betty. They have to travel through magic portals, which is somewhat expensive even for royal families. They can simultaneously use a pair of the portals to move to a different pair of cities, or just one of them can use a portal, while the other one stays where he or she is. At any moment of their travel they have to be in a neighboring cities. They cannot simultaneously move through the same portal.

Write a program that helps Albert and Betty travel from one pair of the cities to another pair. It has to find the cheapest travel plan — with the minimal number of times they have to move though the magic portals. When they move through the portals simultaneously it counts as two moves.

Input

The first line of the input file contains integer numbers n, m, a_1, b_1, a_2, b_2. Here n ($3 \le n \le 100$) is a number of cities in the Kingdom (cities are numbered from 1 to n); m ($2 \le m \le 1000$) is a number of magic portals; a_1, b_1 ($1 \le a_1, b_1 \le n$, $a_1 \ne b_1$) are the neighboring cities where Albert and Betty correspondingly start their travel from; a_2, b_2 ($1 \le a_2, b_2 \le n$, $a_2 \ne b_2$) are the neighboring cities where Albert and Betty correspondingly want to get to ($a_1 \ne a_2$ or $b_1 \ne b_2$).

Following m lines describe the portals. Each line contains two numbers p_{i1} and p_{i2} ($1 \le p_{i1}, p_{i2} \le n$, $p_{i1} \ne p_{i2}$) — cities that are connected by the portal. There is at most one portal connecting two cities.

Output

On the first line of the output file write two numbers c and k. Here c is the minimal number of moves in the travel plan; k is the number of neighboring city pairs that Albert and Betty visit during their travel including a_1, b_1 at the start and a_2, b_2 at the end.

Then write k lines with two integer numbers a'_i and b'_i on each line — consecutive different pairs of neighboring cities that Albert and Betty visit during their travel. If there are multiple travel plans with the same number of moves, then write any of them. It is guaranteed that solution exists.

Sample input and output

kingdom.in	kingdom.out
4 5 1 2 2 1 1 2 2 3 3 4 4 1 1 3	3 3 1 2 2 3 2 1

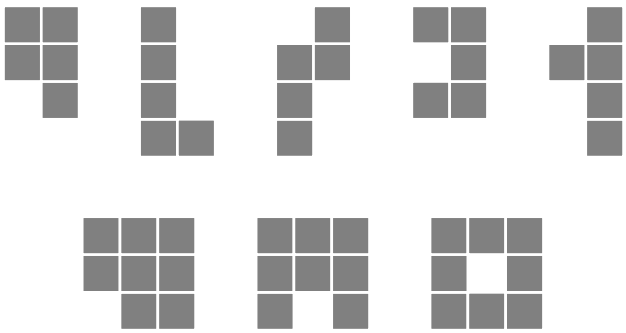
Problem L. Lattice Animals

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

Lattice animal is a set of connected sites on a lattice. Lattice animals on a square lattice are especially popular subject of study and are also known as *polyominoes*. Polyomino is usually represented as a set of sidewise connected squares. Polyomino with n squares is called n-polyomino.

In this problem you are to find a number of distinct *free* n-polyominoes that fit into rectangle $w \times h$. Free polyominoes can be rotated and flipped over, so that their rotations and mirror images are considered to be the same.

For example, there are 5 different pentominoes (5-polyominoes) that fit into 2×4 rectangle and 3 different octominoes (8-polyominoes) that fit into 3×3 rectangle.



Input

The input file consists of a single line with 3 integer numbers n, w, and h ($1 \le n \le 10$, $1 \le w, h \le n$).

Output

Write to the output file a single integer number — the number of distinct free n-polyominoes that fit into rectangle $w \times h$.

Sample input and output

lattice.in	lattice.out
5 1 4	0
5 2 4	5
5 3 4	11
5 5 5	12
8 3 3	3