Problem A. A Pivotal Question

Source file name:	Apivotal.c, Apivotal.cpp, Apivotal.java, Apivotal.py
Input:	`Standard`
Output:	`Standard`

Quicksort is a recursive sorting algorithm developed in 1959 by Tony Hoare. One of the major steps in the algorithm is the *partition* step: given an element p in the array (the *pivot* element) rearrange the elements in the array as shown below where all the values in X_L are $\leq p$ and all elements in X_R are $> p$.

X_L	p	X_R

Figure 1 below shows an array before and after it's been partitioned with the pivot element 13. Note that the elements in X_L and X_R are typically not in sorted order and either one of them could be empty.

13	94	99	1	53	20	63	8	79	11	$\longrightarrow$	1	11	8	13	53	20	63	99	79	94

Figure 1: An array before and after a partition

How a partition is executed and how a pivot element is selected are fascinating questions but are not of interest to us. What we would like you to do is the following: given an array, determine all the values that could be the pivot value assuming the array has been partitioned, or determine that the array has not been partitioned.

Input

Input starts with a positive integer n ($1 \leq n \leq 10^6$) denoting the size of the array. Following this are n positive integers indicating the values in the array. All values are unique and $\leq 10^6$.

Output

Output m = the number of values in the array that could have served as pivot values to partition the array, followed by the pivot values in the order that they appear in the input. If $m > 100$ just output the first 100 of these pivot values. Note that a value of $m = 0$ indicates that the array is not partitioned.

Example

Input	Output
`10 1 11 8 13 53 20 63 99 79 94`	`3 1 13 63`

Problem B. B Road Band

Source file name:	Broad.c, Broad.cpp, Broad.java, Broad.py
Input:	`Standard`
Output:	`Standard`

All the residents of the rural community of Axes Point live on one of two parallel streets separated by a band of green park land. Recently, the local board of supervisors received a grant to (finally) bring wireless service to the town. The grant provides enough money for them to install k access points, and the supervisors have decided to place them in a straight line on County Road "B," which lies in the wooded band midway between the two residential streets. They want to place them in a way that minimizes the distance between users and their nearest access point. Specifically, they want to minimize the sum of the squares of the distances of each user from their nearest access point. For instance, Figure 2 shows two streets with eight customers and their locations along the streets (this is the first sample input). The streets are 3 units apart, and two access points have been placed at points midway between the two streets so that the sum of the eight squared distances is minimized.

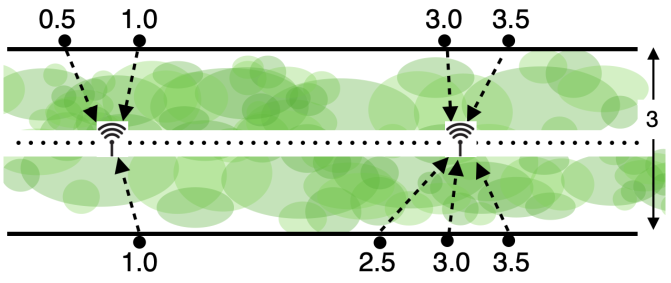

Figure 2: Example Input 1 showing placement of access points

Given the locations of all customers along each of the two streets, the distance between the streets, and the number of access points, help the local government determine the minimum sum of squared distances that can be achieved.

Input

There are three lines of input. The first line contains four integers m, n, k, s, where m and n $(1 \leq m, n \leq 1\,000)$ are the number of customers along each of the two roads, k $(1 \leq k \leq \min(\max(m, n), 100))$ is the number of access points to be placed, and s $(1 \leq s \leq 50)$ is the distance separating the two roads. The second line contains m floating-point values $x_1, x_2, \cdots, x_m$ $(0 \leq x_i \leq 1\,000)$ giving the locations of the m customers along the first road. The third line is similar, containing n floating-point locations of customers along the second road. All values on each of the second and third lines will be distinct (but some values may appear in both lines). Customer locations will have no more than four decimal places.

Output

Output a single floating-point value equal to the minimum sum of squared distances for each customer from the closest of the k access points. Answers should be correct to within an absolute or relative error of 10^{-5}.

Example

Input	Output

`4 4 2 3`
`0.5 1.0 3.0 3.5`
`1.0 2.5 3.0 3.5`
	`18.86666667`

Problem C. Convex Hull Extension

Source file name:	Convex.c, Convex.cpp, Convex.java, Convex.py
Input:	`Standard`
Output:	`Standard`

Dr. Hugh Klidd is a geometry expert who has recently become preoccupied with convex hulls. Recall that for a set of points in the x-y plane, the convex hull is the smallest convex polygon containing all of those points. (A convex polygon has the property that for any two points on/in the polygon, the line segment connecting those two points lies entirely on/in the polygon.) Dr. Klidd has just computed the convex hull of a set of points, S, which he denotes $H(S)$, and is quite pleased with the result:

- the convex hull has $n \geq 3$ vertices
- each vertex has integer coordinates
- no three of the convex hull vertices are collinear, i.e., lie on the same line

Dr. Klidd is ambitious, though, so he wants this convex hull to grow. Specifically, he is looking for an *extension point*, which is a point $p = (x, y)$ satisfying the following conditions:

1. x and y are integers
2. if $S' = S \cup \{p\}$ (S with p added), then the convex hull of S', i.e., $H(S')$, has $n + 1$ vertices
3. no three of these $n + 1$ vertices are collinear

In other words, an extension point increases the number of convex hull vertices by 1, while still keeping all its nice properties. For most convex hulls, $H(S)$, Dr. Klidd can usually find at least one extension point, but he would like to know how many extension points there are to choose from. He postulates that there is an efficient way to count the number of extension points, but having never taken an algorithms course, he turns to you for help.[1]

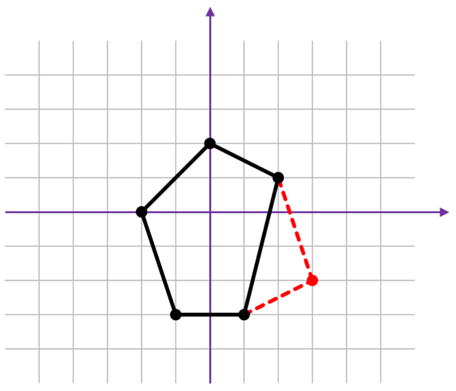

Figure 3: Illustration of an extension point for Example Input 1

Input

The first line of input contains an integer, n, the number of vertices of the convex hull that Dr. Klidd starts with ($3 \leq n \leq 50$). This is followed by n lines, each of which contains two space-separated integers, the x and y coordinates of one of the n vertices ($-1\,000 \leq x, y \leq 1\,000$). The n points are distinct, no three are collinear, and they are given in counterclockwise order.

[1]Dr. Klidd has postulated exactly four things before now, so this is his fifth postulate.

Output

If the number of extension points for the convex hull specified in the input is infinite, output “`infinitely many`”. Otherwise, output the number of extension points.

Example

Input	Output

5
0 2
-2 0
-1 -3
1 -3
2 1
	23

4
-7 -7
7 -7
7 7
-7 7
	infinitely many

Problem D. Cornhusker

Source file name:	Cornhusker.c, Cornhusker.cpp, Cornhusker.java, Cornhusker.py
Input:	`Standard`
Output:	`Standard`

Corn farmers need to do pre-harvest yield estimates to determine the approximate number of bushels of corn their farm will produce. They do this to determine if they have enough storage space (grain bins) to store the harvested crop or if they'll have to store the corn elsewhere, like a co-op (which costs $$$). They also use these estimates when negotiating the future market prices of their corn. Estimates are typically done about a month or two before harvest. By this time, the ears have formed and the kernels on the ears are mostly developed (this makes counting the kernels easier).

According to the *University of Nebraska-Lincoln*, *Nebraska Extension for Educators*, the standard way to estimate corn yield is to calculate the number of bushels of corn per acre. To make the calculations easier, they use an area of 1/1000th of an acre, which, with 30" row spacing, is a section of one row about 17'5" long. Within that 17'5", five ears are chosen at random. For each ear, the number of kernels are counted by multiplying the number of rows of kernels around by the number of kernels over the length of the ear. The totals for each of the five ears are added together and then divided by five to determine the average number of kernels per ear of corn. This number is then multiplied by the total number of ears of corn in the 17'5" section of row. This gives you the total number of kernels in 1/1000th of an acre. This number is then divided by the *Kernel Weight Factor* (*KWF*). The *KWF* is a function of how wet (or dry) the growing season is and is typically a value between 75 (wet) and 95 (dry). The resulting quotient is the number of bushels/acre the farmer can expect to harvest.

For example, suppose that the average number of kernels per ear is 512 (16 kernels around by 32 kernels lengthwise), and there are 25 ears in the 17'5" of row with a *KWF* of 85. The farmer could then expect:

$$\frac{25\times512}{85} = 150\ bushels$$

Since farmers are quite conservative in their estimates, all calculations are done as integers with no rounding.

Input

Input consists of two lines. The first line contains 10 space separated integer values representing the number of kernels around (A) and number of kernels long (L) for each of five ears of corn ($8 \le A \le 24$), ($20 \le L \le 50$).

The second line contains 2 space separated integer values representing the number of ears of corn, N, in the 17'5" row ($10 \le N \le 50$) and the KWF ($75 \le KWF \le 95$).

Output

Output a single integer equal to the estimated number of bushels of corn per acre the farmer can expect given the input supplied.

Example

Input	Output

`16 32 16 32 16 32 16 32 16 32`
`25 85`
	`150`

`14 30 15 32 15 34 14 34 16 32`
`27 75`
	`172`

`16 24 16 34 16 40 14 30 16 35`
`28 95`
	`150`

Problem E. Prof. Fumblemore and the Collatz Conjecture

Source file name:	Fumblemore.c, Fumblemore.cpp, Fumblemore.java, Fumblemore.py
Input:	`Standard`
Output:	`Standard`

The ***Collatz function***, C(n), on positive integers is:

$$n/2 \text{ if } n \text{ is even and } 3n{+}1 \text{ if } n \text{ is odd}$$

The ***Collatz sequence***, CS(n), of a positive integer, n, is the sequence

$$\text{CS}(n) = \text{n, C}(n)\text{, C(C}(n)\text{), C(C(C}(n)\text{)), }\ldots$$

For example, CS(12) = 12, 6, 3, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, . . .

The ***Collatz Conjecture*** (also known as the *3n+1 problem*) is that CS(n) for every positive integer n eventually ends repeating the sequence 4, 2, 1. To date, the status of this conjecture is still unknown. No proof has been given and no counterexample has been found up to very large values.

Prof. Fumblemore wants to study the problem using *Collatz sequence types.* The *Collatz sequence type* (CST) of an integer n, CST(n) is a sequence of letters E and O (for even and odd) which describe the parity of the values in CS(n) up to but not including the first power of 2. So,

$$\text{CST}(12) = \text{EEOEO}$$

Note that

$$\text{CS}(908) = 908,\ 454,\ 227,\ 682,\ 341,\ 1024,\ 512,\ 256,\ 128,\ 64,\ 32,\ 16,\ 8,\ 4,\ 3,\ 2,\ \ldots$$

so 12 and 908 have the same CST.

Prof. Fumblemore needs a program which allows him to enter a sequence of E's and O's and returns the **smallest** integer n for which the given sequence is CST(n).

Notes:

- E's are even numbers which are not powers of 2,
- O's are odd numbers greater than 1.
- The last letter in a sequence must be an O (if C(n) is a power of 2, so is n)
- There can not be two O's in succession (C(odd) = even)
- Since, Prof. Fumblemore does not type well, you must check that the input sequence is valid before attempting to find n. That is, the sequence contains only E's and O's, ends in O and no two O's are adjacent.

Input

Input consists of one line containing a string of up to 50 letters composed of E's and O's.

Output

There is one line of output that consists of the string `INVALID` if the input line is invalid, or a single decimal integer, n, such that n is the *smallest* integer for which CST(n) is the input sequence. Input will be chosen such that $n \leq 2^{47}$.

Example

Input	Output
EEOEO	12
EEOOEO	INVALID

Problem F. Double Up

Source file name:	Doubleup.c, Doubleup.cpp, Doubleup.java, Doubleup.py
Input:	`Standard`
Output:	`Standard`

A Double Up game consists of a sequence of n numbers $a_1, \ldots, a_n$, where each a_i is a power of two. In one move one can either remove one of the numbers, or merge two identical adjacent numbers into a single number of twice the value. For example, for sequence $4, 2, 2, 1, 8$, we can merge the 2s and obtain $4, 4, 1, 8$, then merge the 4s and obtain $8, 1, 8$, then remove the 1, and, finally, merge the 8s, obtaining a single final number, 16. We play the game until a single number remains. What is the largest number we can obtain?

Input

The input consists of two lines. The first line contains n ($1 \leq n \leq 1000$). The second line contains numbers $a_1, \ldots, a_n$, where $1 \leq a_i \leq 2^{100}$ for each i.

Output

The ouput consists of a single line containing the largest number that can be obtained from the input sequence $a_1, \ldots, a_n$.

Example

Input	Output
5 4 2 2 1 8	16

Problem G. Forest for the Trees

Source file name:	Forest.c, Forest.cpp, Forest.java, Forest.py
Input:	`Standard`
Output:	`Standard`

You have sent a robot out into the forest, and it has gotten lost. It has a sensor that will detect all the trees around itself regardless of any occlusions, but unfortunately in this forest, all trees look alike. You do have a map of all trees in the forest, represented as (x, y) points. Conveniently, since this used to be a tree farm, all trees are at integer coordinates, though not all coordinates are occupied. The robot's sensor tells you the x and y distance to each tree within range, relative to the front of the robot. However, the robot is heading in an unknown direction relative to the map, so each sensor reading is given as a tuple of (distance to the right of the robot, distance forward of the robot) and either value can be negative since the robot can sense in all directions. Helpfully, the robot will always place itself at integer coordinates and aligned to the positive or negative x or y axis, and will never be at the same location as a tree. Can you find out where the robot is?

Input

The first line of input contains three integers: n_t, the number of trees in the forest, n_s, the number of trees sensed by the robot, and r_{max}, the maximum Manhattan distance (sum of x and y distances) of any sensor reading. The next n_t lines each contain two integers representing the (x, y) locations of all the trees relative to a global coordinate system. The final n_s lines each contain two integers. The first integer in the i^{th} sensor reading, $s_{i,x}$, represents the distance to the tree along the axis perpendicular to the robot's heading and the second integer $s_{i,y}$ represents the distance along the axis parallel to the robot's heading. You can assume that $|s_{i,x}| + |s_{i,y}| \leq r_{max}$ for all i. You may also assume $0 < n_t \leq 5000$, $0 < n_s \leq 1000$, $0 < r_{max} \leq 1000$, and all tree locations have x and y coordinates $-100,000 \leq x, y \leq 100,000$.

Output

Print one of the following: the x, y location of the robot, printed as two integers separated by a space; "Impossible" if there is no location in the map that could produce the given sensor values, or "Ambiguous" if two or more distinct locations and/or orientations could produce the given sensor values.

Example

Input	Output

4 4 100
1 1
2 2
2 1
3 3
0 1
0 2
-1 2
-2 3
	0 1

Problem H. Impartial Strings

Source file name:	Impartial.c, Impartial.cpp, Impartial.java, Impartial.py
Input:	`Standard`
Output:	`Standard`

Alice builds machines that generates strings. Alice's machines each consist of N states, numbered from 1 to N, and a set of directed edges between these states, each labelled with a character from a fixed set. A subset of the states are "final" states. The machine generates strings by starting at state 1, following a path that terminates at a final state, and concatenating the characters of the edge labels together in the order that the edges are traversed. The path is allowed to visit the same state more than once and can traverse the same edge more than once. The path can pass through final states before eventually terminating at a final state. Self loops are allowed and having two or more edges to and/or from the same state labelled with the same letter is also allowed.

Bob has a favorite string S. Carol has a favorite string T. Alice wonders if she can build a machine that can generate exactly the strings that have an equal number of occurrences of S and T as substrings. That is, the machine should generate every string that has an equal number of occurrences of S and T as substrings and it should not generate any strings that do not satisfy this property. Occurrences may overlap. For example, the string `banana` has two occurrences of the substring `ana`. Help Alice determine if it is possible to complete the task for Bob and Carol's favorite strings.

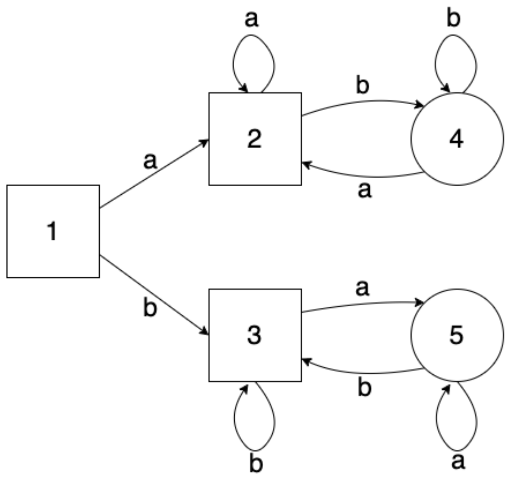

Figure 4: Example machine for the first case in the example input.

Figure 4 gives an example machine for the first case in the example input. The square states represent the final states.

Input

The first line of input contains a single positive integer K ($1 \leq K \leq 50$), the number of test cases. This is followed by K lines each containing three strings A, S, T. The first string, A, is the fixed set of characters used in the machine. The characters in A are distinct lowercase english letters. The second string, S, is Bob's favorite string. The third string, T, is Carol's favorite string. The lengths of S and T satisfy $1 \leq |S|, |T| \leq 500$. It is guaranteed that the distinct characters in S and T are a subset of those in A.

Output

Output one line for each test case. Output 1 if Alice can build a machine as described. Otherwise, output 0.

Example

Input	Output
3	1
ab ab ba	0
abc ab ba	0
cz cczz zzcc	

Problem I. ISBN Conversion

Source file name:	ISBN.c, ISBN.cpp, ISBN.java, ISBN.py
Input:	`Standard`
Output:	`Standard`

ISBN 3-540-42580-2

9 783540 425809

Barcode with ISBN-10 above and ISBN-13 below

An ISBN (International Standard Book Number) is a unique identifier assigned to a distinct edition/version of a published book (for example, hardcover and paperback versions of the same edition get different ISBNs). ISBNs assigned before 2007 were 10 digits long (the ISBN-10 standard), and ISBNs assigned on or after January 1, 2007, are 13 digits long (the ISBN-13 standard). Some books issued before 2007 that are still in print have both an original ISBN-10 and a matching ISBN-13, and some newer books are also given both an ISBN-10 and an ISBN-13, the former for backward-compatibility purposes. That "double identity" situation is the basis for this problem, which requires you to convert valid ISBN-10s to their corresponding ISBN-13s.

The last digit of any ISBN is a *checksum digit* that can be used for simple error detection. ISBN-10 and ISBN-13 use different rules for computing/verifying this last digit:

1. ISBN-10: If the 10 digits from left to right are $d_1, d_2, \ldots, d_{10}$ (so d_{10} is the checksum digit), and if

$$S = 10 \cdot d_1 + 9 \cdot d_2 + 8 \cdot d_3 + \ldots + 2 \cdot d_9 + 1 \cdot d_{10}$$

(coefficients decrease from 10 to 1), then S must be a multiple of 11, i.e., $S \equiv 0 \pmod{11}$.

2. ISBN-13: If the 13 digits from left to right are $d_1, d_2, \ldots, d_{13}$ (so d_{13} is the checksum digit), and if

$$S = 1 \cdot d_1 + 3 \cdot d_2 + 1 \cdot d_3 + 3 \cdot d_4 + \ldots + 3 \cdot d_{12} + 1 \cdot d_{13}$$

(odd-indexed digits are multiplied by 1, even-indexed digits are multiplied by 3), then S must be a multiple of 10, i.e., $S \equiv 0 \pmod{10}$.

It is not hard to see that each rule uniquely determines the checksum digit (given the other digits).

X Factor: Note the following small but important detail for ISBN-10 that does not apply to ISBN-13: because the modulus is 11, the value of the checksum digit lies in $\{0, 1, 2, \ldots, 9, 10\}$, and in the special case that the value of the checksum digit is 10, it is written as X so that only one character is required. So, for example, 039428013X is a valid ISBN-10.

Hyphens: Technically an ISBN-10 consists of four parts, one of which is the checksum digit. (The exact rules defining the first three parts are complicated, so we will not deal with them here.) Two adjacent parts can optionally be separated by a hyphen, which means that an ISBN-10 may contain up to three hyphens, but it cannot begin or end with a hyphen, and it cannot contain consecutive hyphens. If there are three hyphens, one must separate the checksum digit from the digit that precedes it (if there are fewer than three hyphens, there may or may not be a hyphen between the checksum digit and the digit that precedes it). So, for the purposes of this problem, the following are valid ISBN-10s:

039428013-X
0-39-428013X
3-540-42580-2
3540425802

And the following are *invalid* ISBN-10s (the first two because of a hyphen-placement error, the last because it fails the checksum test above):

3-540-4258-02
3-540-425802-
0-14-028333-3

How do you convert an ISBN-10 to an ISBN-13? Simply (i) prepend the three digits 978, (ii) remove the old checksum digit, and (iii) append a new checksum digit as determined by the ISBN-13 rule.[2] To keep things simple, maintain the positions of any existing hyphens, and follow the prepended 978 with a hyphen.

Input

The first line of input contains an integer, T ($1 \leq T \leq 25$), the number of (possibly invalid) ISBN-10s to process. This is followed by T lines, each of which contains a nonempty string of length between 10 and 13, inclusive. Each character is either a base-10 digit (`0` to `9`), a hyphen (`-`), or `X` (capital X).

Output

For each test case, if the candidate ISBN-10 is not valid according to the details given above, output a line containing "`invalid`". Otherwise, output a line containing the corresponding ISBN-13.

Example

Input	Output
4 3-540-4258-02 039428013X 3-540-42580-2 0-14-028333-3	invalid 978-0394280134 978-3-540-42580-9 invalid

[2]In general, an ISBN-13 can begin with three digits other than 978, but only 978 can be prepended to an ISBN-10 to form the matching ISBN-13.

Problem J. Pearls

Source file name:	Pearls.c, Pearls.cpp, Pearls.java, Pearls.py
Input:	`Standard`
Output:	`Standard`

Nikoli's Jewelry Store in Puzzletown sells a line of necklaces consisting of black and white pearls. The pearls in the necklace are firmly glued to a cord of length k, where each unit of cord length either holds a pearl or is empty. Each necklace is displayed on a rectangular velvet-lined surface divided into a grid, where each cell of the grid either holds a pearl, or contains a unit of empty cord, or is unoccupied by either pearl or cord. All cord sections are either horizontal or vertical. A properly-displayed necklace corresponds to a closed, non-self-intersecting path connecting some of the cells of the display.

Because this is, after all, Puzzletown, Nikoli uses some tricky rules governing how the necklace is to be displayed, namely, the rules of a puzzle called "Masyu." When the the necklace is set down along the path (the spacing units on the string match the spacing of the cells on the display surface), the pearls satisfy the constraints of the Masyu puzzle, i.e.,

- A white pearl may not be set down on a cell containing a path corner; in addition, at least one of the two adjacent cells that extend the path through the pearl must contain a corner.
- A black pearl must be placed in a cell containing a path corner; in addition, neither of the two cells extending the path through the black pearl may contain a corner.

An example of a necklace correctly displayed is shown in Figure 5 (this also corresponds to example input 1).

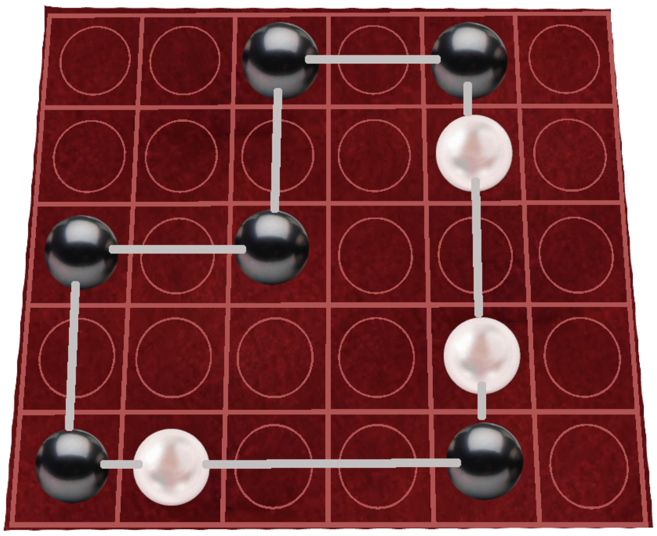

Figure 5: A necklace of length 16 on its display platform

Nikoli's clientele are somewhat picky, so he places three further restrictions on his necklaces. At least half of the necklace's length consists of pearls rather than empty sections of cord. And because black pearls are more desirable (or at least, more expensive) than white ones, the wealthy residents of Puzzletown insist that there be at least twice as many black pearls as white ones. Finally, no two pearls are ever separated by a gap of empty cord longer than five units.

Nikoli sometimes finds that once he has created a necklace according to these restrictions, he is not able to display it according to the rules above. Please help him!

Input

The first line contains 3 integers k, n, and m, where k ($5 \le k \le 60$) is the length of the cord and n and m ($5 \le n, m \le 50$) are respectively the number of rows and columns of the velvet grid. The upper-left cell is row 1, column 1. The second line contains a string of length k consisting only of the characters '`B`', '`W`', and '`.`' (for black pearl, white pearl, and empty cord segment). The first character will always be a pearl—either `B` or `W`. The third line contains two integers r and c ($1 \le r \le n$, $1 \le c \le m$), the row and column of the grid that contains the first pearl in the string.

Output

If there exists a proper way to display the necklace within the given grid boundaries, print a path description of the necklace layout, assuming the first pearl in the string is located at row r, column c of the grid and the path describes the pearls and empty spaces in the same sequence as the input string. The path description should consist of the letters `N`,`S`,`E`, and `W`, indicating whether the path proceeds north, south, east, or west from the current cell. The path should be closed and should not intersect itself. If there is more than one such path, output the one whose description is alphabetically the smallest.

If there is no possible path satisfying the Masyu constraints, output `impossible`

Example

Input	Output

`16 5 6`
`B.B.B.BW.WB..WB.`
`3 1`
	`EENNEESSSSWWWWNN`

`6 5 5`
`W..B.B`
`3 3`
	`impossible`

Problem K. Split Decisions

Source file name:	Split.c, Split.cpp, Split.java, Split.py
Input:	`Standard`
Output:	`Standard`

A Split Decisions puzzle is a type of crossword in which each across and down answer is a pair of words instead of a single word (as in standard crossword puzzles). Each pair of words are identical except in exactly two adjacent positions for which letters have been provided in the puzzle – these sets of letters serve as the clue for the pair of words. In the left half of Figure 6, the letter pairs "IN" and "CR" are given, and the words "SINEW" and "SCREW" can be used to solve that clue. The right half of the figure shows a typical Split Decisions puzzle.

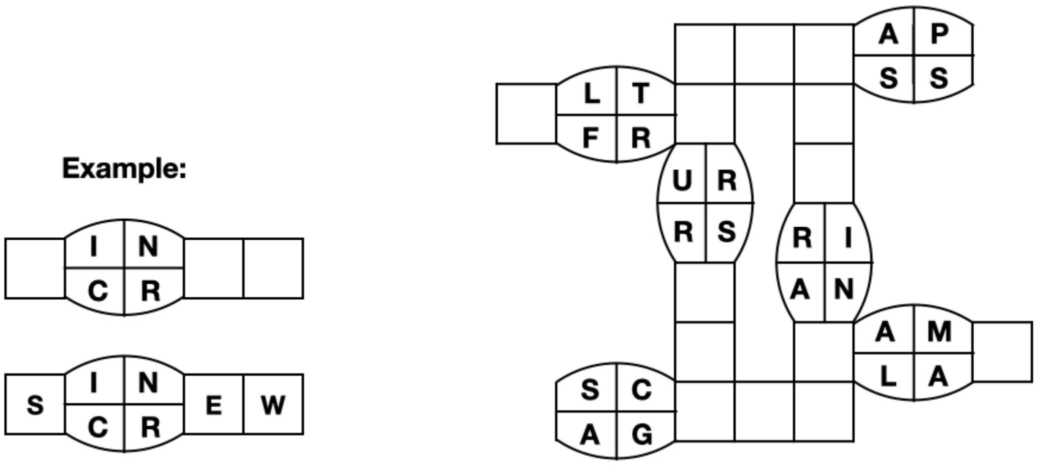

Figure 6: A small Split Decisions puzzle, with an example solved clue.

While we could ask you to solve a given Split Decisions puzzle, we've decided on a slightly different task for you. Given a list of words, we want you to find pairs of words which could be used in a Split Decisions puzzle. Specifically, each pair must differ in **exactly** two consecutive letters AND be the only possible pair of words that work for those sets of letters (i.e., no other pair of words in the list of words solve that clue).

For example, if your list of words contained the words CELL, GULL, GUSH, HALL, and HASH, you could make two clues: `[CE/GU]_ _` (for the answer pair CELL/GULL) and `[CE/HA]_ _` (for the answer pair CELL/HALL). Note that the clue `[GU/HA]_ _` has two possible solutions (GULL/HALL and GUSH/HASH) so it should not be considered.

Input

The first line contains a positive integer n ($1 \leq n \leq 1\,500$) indicating the number of words in the word list. This is followed by n lines each containing one word. Each word consists of uppercase letters and no word has less than 3 or more than 20 letters.

Output

The number of unique pairs of words in the input for which a valid Split Decisions clue exists.

Example

Input	Output
5 CELL GULL GUSH HALL HASH	2

Problem L. A (Fast) Walk in the Woods

Source file name:	Walk.c, Walk.cpp, Walk.java, Walk.py
Input:	`Standard`
Output:	`Standard`

Brice Bilson loves to take jogs in a nearby forest known as Orthogonal Woods. The forest gets that name as the paths – all two-way – are laid out along an orthogonal grid, with all turns being 90 degrees. Brice is a bit persnickety when it comes to his jogs, and always follows a set of rules when he reaches an intersection of two or more paths. These rules are

1. If there are three remaining branches, Brice takes the middle one.
2. If there are just two remaining branches, Brice takes the one on his left.
3. If there are no branches to take, Brice ends his jog and walks to the nearest exit.

Brice is persnickety in another way too. He has assigned each path an "interest value", which is a positive integer indicating how interesting that path is to jog. The higher the value, the more interesting the path is. If the value of a path is n, then Brice will jog on that path no more than n times in his jog. After the n^{th} pass that path will cease to exist as far as Brice is concerned (so, for example, any three-branch intersection using that path now becomes a two-branch intersection and any two-branch intersection becomes a one-branch intersection). An example is shown in Figure 7 below:

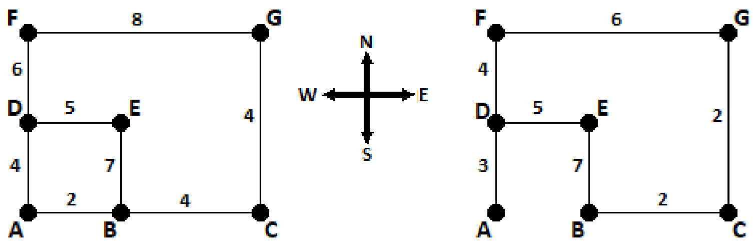

Figure 7: Sample park corresponding to Example Input 1

Suppose Brice enters the park at intersection D heading north in the figure on the left, where the numbers next to each path indicate his interest levels. His travels takes him on the route DFGCBADFGCBA at which point we reach the figure on the right, showing the updated interest levels of each path and the "removal" of the path from A to B since it's now been traversed 2 times. From intersection A Brice now traverses the route ADFGCBEDA at which point he hits a dead end and ends his jog.

Input

Input starts with two integers n and m ($2 \leq n \leq 2\,500$) giving the number of intersections and the number of paths between intersections. The next line contains n pairs of integers giving the locations of the intersections. Intersections are numbered from 1 to n in the order they are presented and all location values x, y satisfy $0 \leq x, y \leq 10^6$. After this are m lines each containing three integers i j k ($1 \leq i, j \leq n, 1 \leq k \leq 10^6$) indicating that a path exists between intersections i and j with interest level k. All paths will be either vertical or horizontal and will not touch any other vertices other than the specified intersection points. The final line of input contains an integer s ($1 \leq s \leq n$) and a character $d \in \{\text{N,S,E,W}\}$ indicating that Brice starts his jog by taking the path in direction d from intersection s. There will always be a path heading in direction d from vertex s.

Output

Output the location where Brice ends his jog.

Example

Input	Output
7 8 0 0 5 0 12 0 0 5 5 5 0 10 12 10 1 2 2 2 3 4 4 5 5 6 7 8 1 4 4 2 5 7 3 7 4 4 6 6 4 N	0 0