Problem A. Airports

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

An airline company offers flights out of n airports, conveniently labeled from 1 to n. The flight time t_{ij} from airport i to airport j is known for every i and j. It may be the case that $t_{ij} \neq t_{ji}$, due to things like wind or geography. Upon landing at a given airport, a plane must be inspected before it can be flown again. This inspection time p_i is dependent only on the airport at which the inspection is taking place and not where the previous flight may have originated.

Given a set of m flights that the airline company must provide, determine the minimum number of planes that the company needs to purchase. The airline may add unscheduled flights to move the airplanes around if that would reduce the total number of planes needed.

Input

The First line of input contains two space-separated integers n and m ($1 \leq n, m \leq 500$). The next line contains n space-separated integers $p_1, \ldots, p_n$ ($0 \leq p_i \leq 10^6$).

Each of the next n lines contains n space-separated integers. The j-th integer in line $i + 2$ is t_{ij} ($0 \leq t_{ij} \leq 10^6$). It is guaranteed that $t_{ii} = 0$ for all i. However, it may be the case that $t_{ij} \neq t_{ji}$ when $i \neq j$.

Each of the next m lines contains three space-separated integers, s_i, f_i and t_i ($1 \leq s_i,\ f_i \leq n$, $s_i \neq f_i$, $1 \leq t_i \leq 10^6$), indicating that the airline company must provide a flight that flies out from airport s_i at exactly time t_i, heading directly to airport f_i.

Output

Print, on a single line, a single integer indicating the minimum number of planes the airline company must purchase in order to provide the m requested flights.

Example

Input	Output
2 2 1 1 0 1 1 0 1 2 1 2 1 1	2
2 2 1 1 0 1 1 0 1 2 1 2 1 3	1
5 5 72 54 71 94 23 0 443 912 226 714 18 0 776 347 810 707 60 0 48 923 933 373 881 0 329 39 511 151 364 0 4 2 174 2 1 583 4 3 151 1 4 841 4 3 993	3

Problem B. Butterfly Effect

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

There are several events that are about to happen. Each event has either a positive outcome or a negative outcome, and these outcomes affect the probabilities of the outcomes of subsequent events.

The events occur in the order given in the input. For every event i, there is an associated integer-valued *base value*, which we denote by b_i. To decide the outcome of an event, we roll a fair m-sided die with sides marked 1 through m and add the amount shown on the die to the base value. If the result is strictly positive, then the outcome is *positive*; otherwise (including if the result is zero), the *negative* outcome occurs. If the positive outcome occurs, then we modify the base values of all subsequent events according to a list of modifiers associated with the event. That is, if the outcome of event i is positive, the new base value for event j is $b_j + p_{ij}$, where p_{ij} is the modifier to event j in the case of a positive outcome for event i. If the negative outcome occurs, we do the same but with a different list of modifiers; the base value for event j becomes $b_j + q_{ij}$, where q_{ij} is the associated modifier.

You have the power to intervene in a certain number of events. When you intervene, instead of rolling one die, you roll two dice and then choose the die you prefer. For each event, you decide whether or not to intervene immediately before that event's die is rolled, i.e., you may use the outcomes of previous events to decide whether or not to intervene. Can you maximize the probability of the final event having a positive outcome?

Input

The first line contains three space-separated integers n, k, and m ($1 \leq k \leq n \leq 20$, $4 \leq m \leq 10^3$), denoting the number of events, the maximum number of interventions, and the die size, respectively. Next are $3n$ lines describing the base values and modifiers of the events, in the following format:

- Line $3i-1$: One integer b_i denoting the base value of event i. The base value of each event will have absolute value at most $2 * 10^3$.
- Line $3i$: $n-i$ space-separated integers $p_{i,i+1}, \ldots, p_{in}$ denoting the modifiers to the base values of events $i+1$ through n in the case of a positive outcome for event i. Each modifier will have absolute value at most $2 * 10^3$.
- Line $3i+1$: $n-i$ space-separated integers $q_{i,i+1}, \ldots, q_{in}$ denoting the modifiers to the base values of event $i+1$ through n in the case of a negative outcome for event i. Each modifier will have absolute value at most $2 * 10^3$.

The final event has no modifiers, and this the last two lines of the input are empty.

Output

Print, on a single line, a single number equal to the maximum probability of the final event having a positive outcome, rounded and displayed to exactly 6 decimal places.

Example

Input	Output
2 2 6 -3 -100 100 0	0.750000
4 1 10 -5 -10 9 0 9 -10 0 -10 0 10 0 0 -10 10 0 -10	0.990000

Problem C. Classy

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

In his memoir So, *Anyway* ..., comedian John Cleese writes of the class difference between his father (who was "middle-middle-middle-lower-middle class") and his mother (who was "upper-upper-lower-middle class"). These fine distinctions between classes tend to confuse American readers, so you are to write a program to sort a group of people by their classes to show the true distinctions.

There are three main classes: upper, middle, and lower. Obviously, upper class is the highest, and lower class is the lowest. But there can be distinctions within a class, so upper-upper is a higher class than middle-upper, which is higher than lower-upper. However, all of the upper classes (upper-upper, middle-upper, and lower-upper) are higher than any of the middle classes.

Within a class like middle-upper, there can be further distinctions as well, leading to classes like lower-middle-upper-middle-upper. When comparing classes, once you've reached the lowest level of detail, you should assume that all further classes are the equivalent to the middle level of the previous level of detail. So upper class and middle-upper class are equivalent, as are middle- middle-lower-middle and lower-middle.

Input

The first line of input contains n ($1 \leq n \leq 10^3$), the number of names to follow. Each of the following n lines contains the name of a person (a sequence of 1 or more lowercase letters 'z'–'z'), a colon, a space, and then the class of the person. The class of the person will include one or more modifiers and then the word **class**. The colon, modifiers, and the word **class** will be separated from each other by single spaces. All modifiers are one of **upper, middle**, or **lower**. It is guaranteed that the input is well-formed. Additionally, no two people have the same name. Input lines are no longer than 256 characters.

Output

Print the n names, each on a single line, from highest to lowest class. If two people have equivalent classes, they should be listed in alphabetical order by name.

Example

Input	Output
5 mom: upper upper lower middle class dad: middle middle lower middle class queenelizabeth: upper upper class chair: lower lower class unclebob: middle lower middle class	queenelizabeth mom dad unclebob chair
10 rich: lower upper class mona: upper upper class dave: middle lower class charles: middle class tom: middle class william: lower middle class carl: lower class violet: middle class frank: lower class mary: upper class	mona mary rich charles tom violet william carl dave frank

Problem D. Magic Trick

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

Your friend has come up with a math trick that supposedly will blow your mind. Intrigued, you ask your friend to explain the trick.

First, you generate a random positive integer k between 1 and 100. Then, your friend will give you n operations to execute. An operation consists of one of the four arithmetic operations `ADD`, `SUBTRACT`, `MULTIPLY`, or `DIVIDE`, along with an integer-valued operand x. You are supposed to perform the requested operations in order.

You don't like dealing with fractions or negative numbers though, so if during the process, the operations generate a fraction or a negative number, you will tell your friend that he messed up.

Now, you know the n operations your friend will give. How many of the first 100 positive integers will cause your friend to mess up?

Input

The first line of input contains a single positive integer n ($1 \leq n \leq 10$). Each of the next n lines consists of an operation, followed by an operand. The operation is one of the strings `ADD`, `SUBTRACT`, `MULTIPLY`, or `DIVIDE`. Operands are positive integes not exceeding 5.

Output

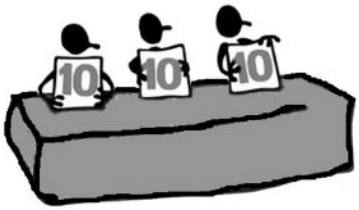
Print, on a single line, a single integer indicating how many of the first 100 positive integers will result in you telling your friend that he messed up.

Example

Input	Output
1 SUBTRACT 5	4
1 DIVIDE 2	50
2 ADD 5 DIVIDE 5	80

Problem E. Excellence

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



The World Coding Federation is setting up a huge online programming tournament of teams comprised of pairs of programmers. Judge David is in charge of putting teams together from the Southeastern delegation. Every student must be placed on exactly one team of two students. Luckily, he has an even number of students who want to compete, so that he can make sure that each student does compete. However, he'd like to maintain his pristine reputation amongst other judges by making sure that each of the teams he fields for the competition meet some minimum total rating. We define the total rating of a team to be the sum of the ratings of both individuals on the team.

Help David determine the maximum value, X, such that he can form teams, each of which have a total rating greater than or equal to X.

Input

The first line of input contains a single positive integer n ($1 \leq n \leq 10^5$, n is even), the number of students who want to enter the online programming tournament. Each of the following n lines contains one single integer s_i ($1 \leq s_i \leq 10^6$), the rating of student i.

Output

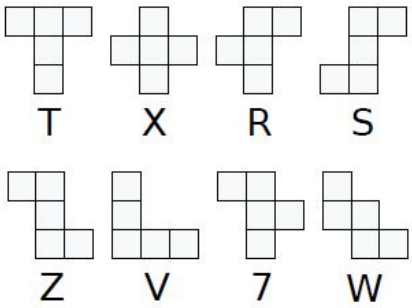
Print, on a single line, the maximum value, X, such that David can form teams where every team has a total rating greater than or equal to X.

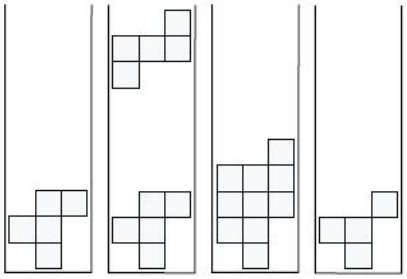
Example

Input	Output
4 1 2 3 5	5
2 18 16	34
4 13 12 19 14	27

Problem F. Falling Blocks

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





FallingBlocks is a Tetris-like arcade game, played on a board with 3 columns and 10 rows according to the following rules. A known, indefinitely repeating sequence of pentominoes (simply called *pieces*) fall down from the top of the board, one at a time. The 8 pieces and their labels are shown above.

The pieces can be rotated freely (by 0, 90, 180 or 270 degrees), but they cannot flipped.

The rules are similar to that of Tetris. The newly introduced piece falls from the top of the board as far as possible until it hits the bottom of the board or an existing block in the board. Then, any rows that are completely full are removed and rows above are moved down, with no further change in the rows themselves.

To illustrate this, consider an empty board, on which an `R` piece, followed by a `Z` piece, falls. If we drop the `R` piece without rotating, we end up with first board shown above. Dropping a rotated `Z` piece on top of it causes two rows to fill. These rows are removed, and the rows above are pushed downward. The final situation is as shown in the rightmost picture. The final position has a "hanging block;" this block does not fall any further at this point.

Unlike Tetris, the top three rows of the board must be completely empty in order to place a piece, i.e., if any of the top three rows is not empty after removing all rows that are full, the game is over.

The score is solely based on the number of pieces played on the board before the game is over. Given the sequence of pieces that repeats indefinitely, determine the maximum number of pieces that can be played.

Below are some example sequences of pieces, followed by explanation.

- `X`: Every drop of an `X` piece leaves two rows filled that cannot be removed by additional `X` pieces. After placing four pieces, we have eight non-empty rows left, so the next `X` piece cannot be placed. So the result is 4.
- `XXXXR`: The `R` piece could be rotated to not overlap the square left in the highest non-empty row, but our rule is that the top three rows must be completely empty to place any piece. So the result is 4.
- `VZV`: Two `V` pieces and a `Z` piece can be placed to clear the board, so this game can go on forever.

Input

The input consists of a single line that contains a single string, representing the sequence of pentominoes. The input sequence contains between 1 and 20 characters.

Output

Print, on a single line, the maximum number of pieces that can be played until no more piece can be placed on the board. If the game can continue indefinitely, print `forever`.

Example

Input	Output
T	forever
W	8
VZ	25
VR	forever
VVTRX	130
XSVTVT	forever

Problem G. Gears

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

A set of gears is installed on the plane. You are given the center coordinate and radius of each gear. For a given input and output gear, indicate what happens to the output gear if you attempt to rotate the input gear.

Input

The first line of input contains a single positive integer n ($2 \leq n \leq 10^3$), the total number of gears. Following this will be n lines, one per gear, containing three space-separated integers x_i, y_i, and r_i ($-10^4 \leq x_i, y_i \leq 10^4$, $1 \leq r_i \leq 10^4$), indicating the center coordinate and the radius of the i-th gear. Assume the tooth count for each gear is sufficiently high that the gears always mesh correctly. It is guaranteed that the gears do not overlap with each other. The input gear is the first gear in the list, and the output gear is the last gear in the list.

Output

If the input gear cannot move, print, on a single line, "`The input gear cannot move.`" (without the quotation marks).

If the input gear can move but is not connected to the output gear, print, on a single line, "`The input gear is not connected to the output gear.`" (without the quotation marks).

Otherwise, print, on a single line, the ratio the output gear rotates with respect to the input gear in the form of "`##:##`" (without the quotation marks), in reduced form. If the output gear rotates in the opposite direction as the input gear, write the ratio as a negative ratio. For example, if the output gear rotates clockwise three times as the input gear rotates counterclockwise twice, the output should be `-3:2`.

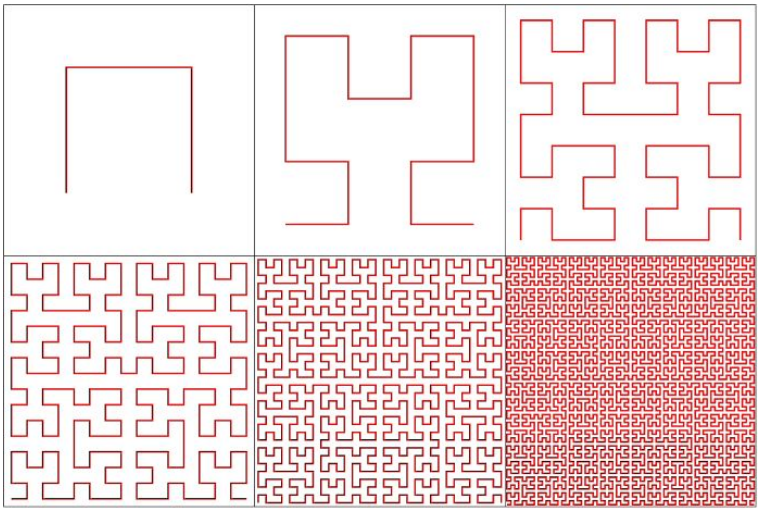
Example

Input	Output
2 0 0 100 200 0 100	-1:1
3 0 0 100 200 0 100 400 0 100	1:1
16 10 10 5 20 10 5 30 10 5 40 10 5 10 20 5 20 20 5 30 20 5 40 20 5 10 30 5 20 30 5 30 30 5 40 30 5 10 40 5 20 40 5 30 40 5 40 40 5	1:1
3 0 0 1 0 3 2 4 0 3	The input gear cannot move.

Problem H. Hilbert Sort

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

In database storage, arranging data items according to a numeric key not only makes it easier to search for a particular item, but also makes better use of a CPU's cache: any segment of data that's contiguous in memory will describe items with similar keys. This is useful if, for instance, we want to access all items whose keys are in some range. Things get more complicated if the keys represent points on a 2D grid, as might happen in a GPS guidance system. If the points $(x,\ y)$ are sorted primarily by x, breaking ties by y, then points that are adjacent in memory will have similar x coordinates but not necessarily similar y, potentially placing them far apart on the grid. To better preserve distances, we may sort the data along a continuous space-filling curve.



We consider one such space-filling curve called the *Hilbert curve*. The Hilbert curve starts at the origin $(0, 0)$ and finishes at $(S, 0)$, in the process traversing the entire axis-aligned square with corners at $(0, 0)$ and (S, S). It has the following recursive construction: split the square into four quadrants meeting at $(S/2, S/2)$, and recursively fill each of them with a suitably rotated and scaled copy of the full Hilbert curve. First, the lower-left quadrant is filled with a curve going from $(0, 0)$ to $(0, S/2)$. Second, the upper-left quadrant is filled from $(0, S/2)$ to $(S/2, S/2)$. Third, the upper- right quadrant is filled from $(S/2, S/2)$ to $(S, S/2)$. And finally, the lower-right quadrant is filled from $(S, S/2)$ to $(S, 0)$. The Hilbert curve can alternatively be constructed as the mathematical limit of a sequence of curves, the first six of which are shown in the figure.

Given some locations of interest, you are asked to sort them according to when the Hilbert curve visits them. Note that while the curve intersects itself at infinitely many places, e.g., at $(S/2, S/2)$; making S odd guarantees that all integer points are visited just once.

Input

The first line of input contains two space-separated integers n and S ($1 \leq n \leq 2 * 10^5,\ 1 \leq S < 10^9, S$ is odd). This is followed by n lines. Line $i + 1$ describes the i-th location of interest by space-separated integers x_i and y_i ($0 \leq x_i,\ yi \leq S$) and an identifier string consisting of at most 46 alphanumeric characters ('A' - 'Z', 'a'-'z', '0'-'9'). No two locations will share the same position or the same identifier.

Output

Print the n identifier strings, one on each line, Hilbert-sorted according to their positions.

Example

Input	Output
14 25 5 5 Honolulu 5 10 PugetSound 5 20 Victoria 10 5 Berkeley 10 10 Portland 10 15 Seattle 10 20 Vancouver 15 5 LasVegas 15 10 Sacramento 15 15 Kelowna 15 20 PrinceGeorge 20 5 Phoenix 20 10 SaltLakeCity 20 20 Calgary	Honolulu Berkeley Portland PugetSound Victoria Vancouver Seattle Kelowna PrinceGeorge Calgary SaltLakeCity Sacramento LasVegas Phoenix

Problem I. Olympics

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

The weightlifting event is up next at the Olympic games, and it's time to impress your fans! To accomplish your sequence of lift attempts, you have a constant strength S and a decreasing energy reserve E. For each attempt, you may choose any positive (not necessarily integer) weight W. If $S \geq W$, the lift succeeds and your energy goes down by E_{succ}. If $S < W$, the lift fails and your energy goes down by E_{fail}. You may continue attempting lifts as long as $E > 0$. If at any point $E \leq 0$, you can make no further attempts. Your score is the maximum weight in kg that you successfully lift, or 0 if all attempts failed.

Ideally, you should lift at exactly your strength limit. However, you do not know your strength. You only know that you can definitely lift the 25 kg Olympic bar, and that the maximum conceivable lift adds 100 kg on each side for a total of 225 kg. How close to an optimal score can you guarantee? That is, what's the smallest d for which you can ensure a score of at least $S - d$?

Input

The input consists of a single line containing three space-separated integers E, E_{succ}, and E_{fail} ($1 \leq E,\ E_{succ},\ E_{fail} \leq 10^7$).

Output

Print, on a single line, the minimum d, rounded and displayed to exactly 6 decimal places.

Example

Input	Output
1 3 3	112.500000
12 3 3	13.333333
3000 2 3	0.000000

Problem J. Triangle

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

Determine if it is possible to produce two triangles of given side lengths, by cutting some rectangle with a single line segment, and freely rotating and flipping the resulting pieces.

Input

The input consists of two lines. The first line contains three space-separated positive integers, indicating the desired side lengths of the first triangle. Similarly, the second line contains three space-separated positive integers, denoting the desired side lengths of the second triangle. It is guaranteed that the side lengths produce valid triangles. All side lengths are less than or equal to 100.

Output

Print, on a single line, whether there exists a rectangle which could have been cut to form triangles of the given side lengths. If such a rectangle exists, print `YES`. Otherwise, print `NO`.

Example

Input	Output
3 4 5 4 3 5	YES
3 4 6 4 6 3	NO
39 52 65 25 60 65	NO