Problem A. Architecture

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

Your brother has won an award at the recent Breakthroughs in Architectural Problems Conference and has been given the once in a lifetime opportunity of redesigning the city center of his favorite city Nijmegen. Since the most striking parts of a city's layout are the skylines, your brother has started by drawing ideas for how he wants the northern and eastern skylines of Nijmegen to look. However, some of his proposals look rather outlandish, and you are starting to wonder whether his designs are possible.

For his design, your brother has put an $R \times C$ grid on the city. Each cell of the city will contain a building of a certain height. The eastern skyline is given by the tallest building in each of the R rows, and the northern skyline is given by the tallest building in each of the C columns.

A pair of your brother's drawings of skylines is possible if and only if there exists some way of assigning building heights to the grid cells such that the resulting skylines match these drawings.

Figure 1 shows a possible city with the northern and eastern skylines exactly as given in the input of the first sample.

1	2	3	4
0	1	2	3
1	2	1	1
1	0	1	1

Example city showing sample 1 has a valid solution.

Input

- The first line consists of two integers $1 \leq R, C \leq 100$, the number of rows and columns in the grid.
- The second line consists of R integers $x_1, \ldots, x_R$ describing the eastern skyline ($0 \leq x_i \leq 1000$ for all i).
- The third line consists of C integers $y_1, \ldots, y_C$ describing the northern skyline ($0 \leq y_j \leq 1000$ for all j).

Output

Output one line containing the string `possible` if there exists a city design that produces the specified skyline, and `impossible` otherwise.

Example

Input	Output

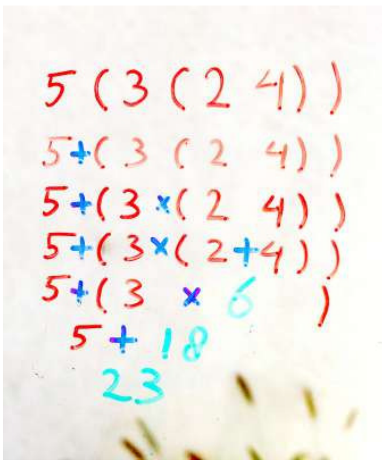
`4 4`
`4 3 2 1`
`1 2 3 4`
	`possible`

`4 4`
`1 2 3 4`
`1 2 3 2`
	`impossible`

Problem B. Bracket Sequence

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

Two great friends, Eddie John and Kris Cross, are attending the Brackets Are Perfection Conference. They wholeheartedly agree with the main message of the conference and they are delighted with all the new things they learn about brackets.



One of these things is a *bracket sequence*. If you want to do a computation with $+$ and $\times$, you usually write it like so:

$$(2 \times (2 + 1 + 0 + 1) \times 1) + 3 + 2.$$

The brackets are only used to group multiplications and additions together. This means that you can remove all the operators, as long as you remember that addition is used for numbers outside any parentheses! A *bracket sequence* can then be shortened to

$$(\ 2\ (\ 2\ 1\ 0\ 1\ )\ 1\ )\ 3\ 2.$$

That is much better, because it saves on writing all those operators. Reading bracket sequences is easy, too. Suppose you have the following bracket sequence

$$5\ 2\ (\ 3\ 1\ (\ 2\ 2\ )\ (\ 3\ 3\ )\ 1\ ).$$

You start with addition, so this is the same as the following:

$$5 + 2 + (\ 3\ 1\ (\ 2\ 2\ )\ (\ 3\ 3\ )\ 1\ ).$$

You know the parentheses group a multiplication, so this is equal to

$$5 + 2 + (3 \times 1 \times (\ 2\ 2\ ) \times (\ 3\ 3\ ) \times 1).$$

Then there is another level of parentheses: that groups an operation within a multiplication, so the operation must be addition.

$$5 + 2 + (3 \times 1 \times (2 + 2) \times (3 + 3) \times 1) = 5 + 2 + (3 \times 1 \times 4 \times 6 \times 1) = 5 + 2 + 72 = 79.$$

Since bracket sequences are so much easier than normal expressions with operators, it should be easy to evaluate some big ones. We will even allow you to write a program to do it for you.

Note that ( ) is not a valid bracket sequence, nor a subsequence of any valid bracket sequence.

Input

- One line containing a single integer $1 \le n \le 3 \cdot 10^5$.
- One line consisting of n tokens, each being either `(`, `)`, or an integer $0 \le x < 10^9+7$. It is guaranteed that the tokens form a bracket sequence.

Output

Output the value of the given bracket sequence. Since this may be very large, you should print it modulo $10^9 + 7$.

Example

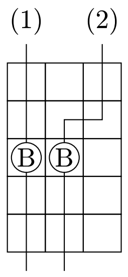
Input	Output
2 2 3	5
8 (2 (2 1)) 3	9
4 (12 3)	36
6 (2) (3)	5
6 ((2 3))	5
11 1 (0 (583920 (2839 82)))	1

Problem C. Canyon crossing

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

The Bridge And Passageway Creators are responsible for making new paths through the local mountains. They have approved your plan to build a new route through your favorite canyon. You feverishly start working on this beautiful new path, when you realize you failed to take into account the flow of a nearby river: the canyon is flooded! Apparently this happens once every blue moon, making some parts of the path inaccessible. Because of this, you want to build a path such that the lowest point on the path is as high as possible. You quickly return to the village and use all of your money to buy rope bridges. You plan to use these to circumvent the lowest parts of the canyon.

1	1	3
3	3	3
0	0	0
2	2	1
1	2	1



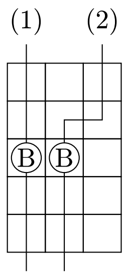
Canyon and two possible paths with minimal height 1 and 2 for sample input 1. The `B` indicate bridges.

Your map of the canyon consists of a rectangular grid of cells, each containing a number giving the height of the terrain at that cell. The path will go from the south side of the canyon (bottom on your map) to the north side (top of your map), moving through a connected sequence of cells. Two cells are considered connected if and only if they share an edge. In particular, two diagonally touching cells are not considered to be connected. This means that for any cell not on the edge of the map, there are 4 other cells connected to it. The left of figure 2 contains the map for the first sample input.

The path through the canyon can start on any of the bottom cells of the grid, and end on any of the cells in the top tow, like the two paths on the right in 2. The lowest height is given by the lowest height of any of the cells the paths goes through. Each bridge can be used to cross exactly one cell. This cell is then not taken into account when calculating the minimal height of the path. Note that is allowed to chain multiple bridges to use them to cross multiple cells,

Given the map of the canyon and the number of bridges available, find the lowest height of an optimal path.

1	1	3
3	3	3
0	0	0
2	2	1
1	2	1



Canyon and an optimal path for sample input 2.

Input

- A single line containing three integers: $1 \leq R \leq 1000$ and $1 \leq C \leq 1000$, the size of the map, and

$0 \leq K \leq R - 1$, the number of bridges you can build.

- This is followed by R lines each containing C integers. The j-th integer on the i-th line corresponds to the height $0 \leq H_{i,j} \leq 10^9$ of the canyon at point (i, j). The first line corresponds to the northern edge of the canyon, the last line to the southern edge.

Output

Output a single integer, the lowest height of the optimal path.

Example

Input	Output

5 3 1
1 1 3
3 3 3
0 0 0
2 2 1
1 2 1
	2

5 3 3
2 1 1
2 1 1
1 1 1
1 1 2
1 1 2
	2

3 2 2
1 1
4 4
1 2
	4

Problem D. Deceptive Dice

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

Recently your town has been infested by swindlers who convince unknowing tourists to play a simple dice game with them for money. The game works as follows: given is an n-sided die, whose sides have $1, 2, \ldots, n$ pips, and a positive integer k. You then roll the die, and then have to make a choice. Option 1 is to stop rolling. Option 2 is to reroll the die, with the limitation that the die can only be rolled k times in total. Your score is the number of pips showing on your final roll.

Obviously the swindlers are better at this game than the tourists are. You, proud supporter of the Battle Against Probabilistic Catastrophes, decide to fight this problem not by banning the swindlers but by arming the tourists with information.

You create pamphlets on which tourists can find the maximum expected score for many values of n and k. You are sure that the swindlers will soon stop their swindling if the tourists are better prepared than they are!

The layout of the flyers is done, and you have distribution channels set up. All that is left to do is to calculate the numbers to put on the pamphlet.

Given the number of sides of the die and the number of times you are allowed to roll, calculate the expected (that is, average) score when the game is played optimally.

Input

- A single line with two integers $1 \leq n \leq 100$, the number of sides of the die, and $1 \leq k \leq 100$, the number of times the die may be rolled.

Output

Output the expected score when playing optimally. Your answer should have an absolute error of at most 10^{-7}.

Example

Input	Output
`1 1`	`1`
`2 3`	`1.875`
`6 2`	`4.25`
`8 9`	`7.268955230712891`

Problem E. #exclude<scoring>

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

You are participating in a programming contest cup. The cup consists of a series of programming contests, followed by a final at the end of the season for the 15 top ranked contestants in the cup. With only one contest left to go before the final, you are starting to wonder if your performance in the earlier contests has been good enough to already secure you a spot in the finals. If so, you could succumb to your laziness and skip the last contest.

The ranking of the cup works as follows. In each contest, a contestant earns some number of points between 0 and 101 (the details of this are described below). Their *aggregate score* is then defined to be the *sum of the four highest scores* achieved. For instance if a contestant got 45, 15, 32, 0, 30, and 20 points over 6 contests, their aggregate score is $45+32+30+20 = 127$. The *rank* of a contestant X *in the cup* is defined to be 1 plus the number of contestants that have a strictly larger aggregate score than X.

The score a contestant earns from a contest is based on the rank they achieve *in that contest*, according to the following table.

Rank	Points	Rank	Points	Rank	Points
1	100	11	24	21	10
2	75	12	22	22	9
3	60	13	20	23	8
4	50	14	18	24	7
5	45	15	16	25	6
6	40	16	15	26	5
7	36	17	14	27	4
8	32	18	13	28	3
9	29	19	12	29	2
10	26	20	11	30	1

If a contestant gets a worse rank than 30, they get 0 points. If two or more contestants get the same rank in the contest, they are instead assigned the average points of all the corresponding ranks. This average is always rounded up to the closest integer. For example, if three contestants are tied for second place they all receive $\lceil \frac{75+60+50}{3} \rceil = 62$ points, and the next contestant will have rank 5 and receives 45 points (or less, if there is a tie also for 5'th place). This applies also at rank 30, e.g., if 4 711 contestants are tied for 30'th place, they all receive 1 point.

Contestants may participate in every contest either on-site or online. If they compete on-site, they get 1 extra point, no matter their original number of points. If a contestant does not participate in a contest, they get 0 points.

Input

The first line of input contains two integers n and m ($2 \le n \le 10$, $1 \le m \le 10^5$), where n is the number of contests in the cup (excluding the final), and m is the number of people who participated in any of the first $n-1$ contests.

Then follow m lines, each describing a contestant. Each such line consists of $n-1$ integers $0 \le s_1, \ldots, s_{n-1} \le 101$, where s_i is the score that this contestant received in the ith contest.

The first contestant listed is you. The point values in the input might not correspond to actual points from a contest.

Output

Output a single integer r, the worst possible rank you might end up in after the last contest, assuming you do not participate in it.

Example

Input	Output
4 2	2
50 50 75	
25 25 25	
5 2	1
50 50 50 50	
25 25 25 25	
2 4	3
90	
1	
3	
2	

Problem F. Factor 2

Source file name:	factor2.c, factor2.cpp, factor2.java, factor2.py
Input:	`Standard`
Output:	`Standard`

Strings of yarn have been popular in Catland for ages. Which cat has not spent many a lazy afternoon bouncing around a ball of yarn? Lately however, strings of yarn have gotten competition: strings of characters. It turns out that these are almost as much fun as yarn, and generally much safer as well (so far, no cat has had to call 911 on account of any character string-related entanglement accidents).

Picture by Stefan Tell on Flickr, cc by

Naturally, some strings are more stylish than others, and for cool cats it is important to engage in their string-playing pastime with style. The *meow factor* of a string S is the minimum number of operations needed to transform S into a string S' which contains the word "meow" as a substring, where an operation is one of the following four:

1. Insert an arbitrary character anywhere into the string.
2. Delete an arbitrary character anywhere from the string.
3. Replace any character in the string by an arbitrary character.
4. Swap any two adjacent characters in the string.

Write a program to compute the meow factor of a string of characters.

Input

The input consists of a single line containing a string S, consisting only of lower-case letters '`a`'-'`z`'. The length of S is at least 1 and at most 10^6.

Output

Output the meow factor of S.

Example

Input	Output
`pastimeofwhimsy`	`1`
`yarn`	`4`

Problem G. Greetings!

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

Now that Snapchat and Slingshot are *soooo* 2018, the teenagers of the world have all switched to the new hot app called BAPC (Bidirectional and Private Communication). This app has some stricter social rules than previous iterations. For example, if someone says goodbye using `Later!`, the other person is expected to reply with `Alligator!`. You can not keep track of all these social conventions and decide to automate any necessary responses, starting with the most important one: the greetings. When your conversational partner opens with `he...ey`, you have to respond with `hee...eey` as well, but using twice as many `e`'s!

heey

heeeey

what's up

what's uhat's up

Later!

Alligator!

Given a string of the form `he...ey` of length at most 1000, print the greeting you will respond with, containing twice as many `e`'s.

Input

- The input consists of one line containing a single string s as specified, of length at least 3 and at most 1000.

Output

Output the required response.

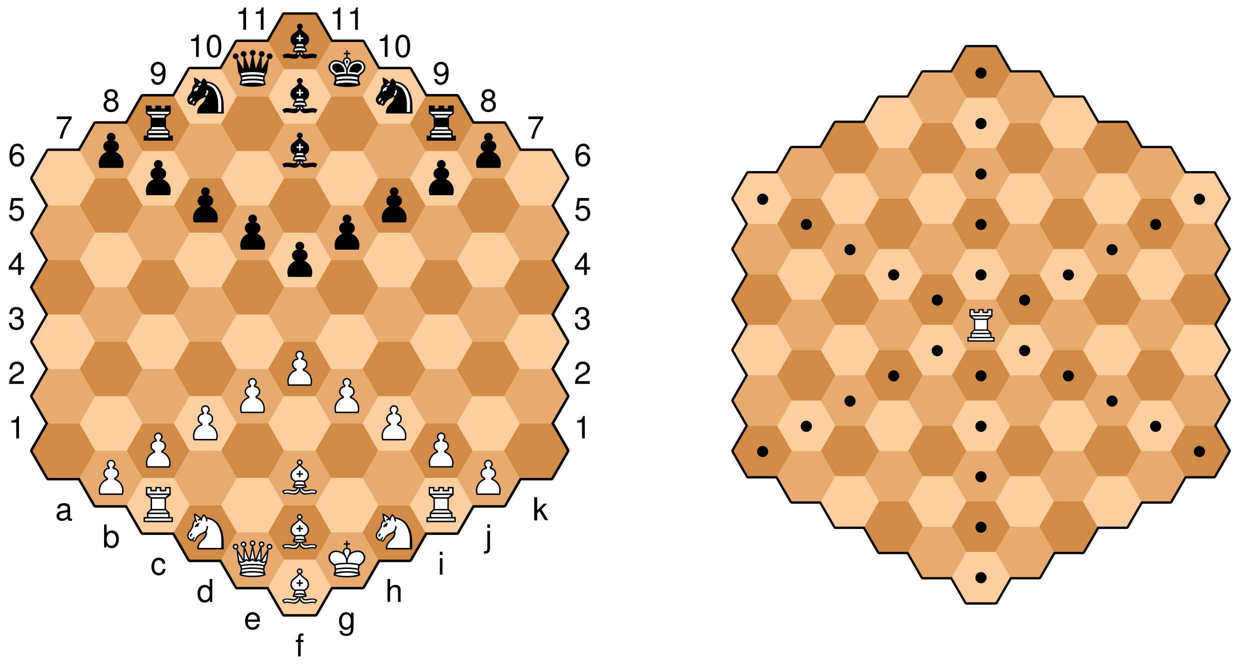
Example

Input	Output
`hey`	`heey`
`heeeeey`	`heeeeeeeeeey`

Problem H. Hexagonal Rooks

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

It is game night and Alice and Bob are playing chess. After beating Bob at chess several times, Alice suggests they should play a chess variant instead called *hexagonal chess.* Although the game is very rarely played nowadays, Alice knows the rules very well and has obtained a hexagonal chessboard from her subscription to the magazine of Bizarre Artifacts for Playing Chess.



The field naming of the hexagonal chess board and the directions in which a rook can move.

The hexagonal chess board, shown above, consists of 91 hexagonal cells arranged in the shape of a hexagon with side length 6 as depicted in the above diagrams. The board is divided into 11 columns, each called a file, and the files are labeled `a` to `k` from left to right. It is also divided into 11 v-shaped rows, each called a rank, which are labeled `1` to `11` from bottom to top. The unique cell in file x and rank y is then denoted by the coordinate xy. For example, rank `11` contains only a single cell `f11` and rank `7` is occupied entirely by the black player's pawns.

Alice begins by explaining how all the pieces move. The simplest piece is the rook, which can move an arbitrary positive number of steps in a straight line in the direction of any of its 6 adjacent cells, as depicted in the figure on the right. Bob immediately realises that the hexagonal rook already is more difficult to work with than its regular chess counterpart.

In order to attack one of the opponents pieces, it is useful to know which cells his rook can move to such that it attacks the opposing piece. The more of these cells there are, the more valuable the current position of his rook is. However, calculating this number is too much for Bob. After losing so many games of regular chess, Alice allows Bob to use a program to assist in his rook placement. While Alice explains the rest of the game you get busy coding.

As a small simplification, Bob will compute the number of ways his rook can move to the destination cell assuming there are no other pieces on the board, not even the piece he wants to attack.

Input

- The input consists of one line, containing two different coordinates on the hexagonal chess board, the current positions of your rook and the piece you want to attack.

Output

Output a single integer, the number of ways the rook can move from its current position to the position of the piece it wants to attack in exactly two moves, assuming there are no other pieces on the board.

Example

Input	Output
c4 h4	6
a1 a2	5

Problem I. Inquiry I

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

The Bureau for Artificial Problems in Competitions wants you to solve the following problem: Given n positive integers $a_1, \ldots, a_n$, what is the maximal value of

$$\left(a_1^2 + \cdots + a_k^2\right) \cdot (a_{k+1} + \cdots + a_n)?$$

Input

- A single line containing an integer $2 \leq n \leq 10^6$.
- Then follow n lines, the ith of which contains the integer $1 \leq a_i \leq 100$.

Output

Output the maximal value of the given expression.

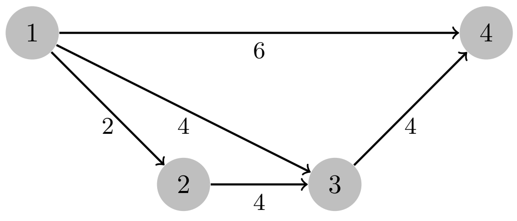
Example

Input	Output
5 2 1 4 3 5	168
2 1 1	1
10 8 5 10 9 1 4 12 6 3 13	10530

Problem J. Jumbled Journey

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

Together with some friends you are planning a holiday to the Beautiful Authentic Parks Centre. While there, you want to visit the parks as much as you can. As part of the preparation you, together with your best friend, decided to make an extensive map of the parks, the roads between them, and the lengths of these roads. To ensure that the visitors to the Parks Centre do not circle endlessly through the gorgeous parks, the routes between the parks are one-way only and are made such that it is impossible to visit a park more than once (that is, the corresponding graph is acyclic). Together you spend the entire day to create an incredibly detailed and admittedly rather large map of the Parks Centre.



The map of the parks corresponding to sample input 1.

The next day disaster strikes: your friend happily announces that he got rid of the cumbersome map! Instead, he decided to replace it with a simple table containing only the average distances between the parks, weighing each route equally. Thus, he would give you an average distance of 8 between park 1 and park 4 as in the previous figure, because there are 3 paths and their average length is $(6 + 8 + 10)/3 = 8$. You feel defeated, and you dread the thought of making the map all over again. Perhaps it might be easier to try and use the table of average distances your friend made in order to reconstruct the original map. One thing you remember, is that the roads on the map were never inefficient. If there was a direct road between two parks, then every path via at least one other park was *strictly* longer. This condition holds for the first sample input because, for example, the road from 1 to 4 has length 6, which is shorter than the length of any other path from 1 to 4.

Given an input of average distances between parks, output the original map: a weighted directed acyclic graph for which these average distances hold.

Input

- The first line contains an integer $1 \leq n \leq 100$, the number of vertices (parks).
- The next n lines contain the average distance from vertex i to vertex j as the jth number on the ith line, or -1 in case there is no path from vertex i to vertex j. All distances are either -1 or non-negative integers, at most 10^{15}.

You know the following facts about the map (the original graph):

- There is no way to follow the roads and return to a park once you have left it. That is, the graph is acyclic.
- The lengths of the roads in the original graph are all integers.
- Between any two parks, there are at most 1000 distinct paths you can take from one to the other.

- Direct roads are always efficient: if there is a direct road from park i to park j, every other path from i to j is strictly longer than that direct road.

Output

Print n lines each containing n integers. The jth number on the ith line should be the positive length of the edge from vertex i to vertex j, or -1 if there is no such edge. It is guaranteed that a solution exists.

Example

Input	Output

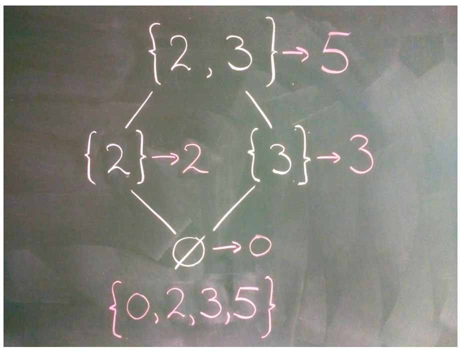
4
0 2 5 8
-1 0 4 8
-1 -1 0 4
-1 -1 -1 0
	-1 2 4 6
	-1 -1 4 -1
	-1 -1 -1 4
	-1 -1 -1 -1

6
0 -1 48 -1 132 -1
24 0 84 36 153 108
-1 -1 0 -1 84 -1
-1 -1 60 0 116 72
-1 -1 -1 -1 0 -1
-1 -1 -1 -1 96 0
	-1 -1 48 -1 -1 -1
	24 -1 -1 36 -1 -1
	-1 -1 -1 -1 84 -1
	-1 -1 60 -1 36 72
	-1 -1 -1 -1 -1 -1
	-1 -1 -1 -1 96 -1

Problem K. Knapsack Packing

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

One of the most difficult things about going on a holiday is making sure your luggage does not exceed the maximum weight. You, chairman of the Backpacker's Association for Packing Carry-ons, are faced with exactly this problem. You are going on a lovely holiday with one of your friends, but now most of your time is spent in frustration while trying to pack your backpack. In order to optimize this process, you and your friend have independently set upon trying to find better ways to pack.



After some time you have a serious breakthrough! Somehow you managed to solve the Knapsack problem in polynomial time, defying expectations everywhere. You are not interested in any theoretical applications, so you immediately return to your friend's apartment in order to now quickly pack your backpack optimally.

When you arrive there, you find that your friend has set upon her own solution, namely to enumerate all possible packings. This means that all items you possibly wanted to bring are scattered across the entire apartment, and it would take a really long time to get all the items back together.

Luckily you can use the work your friend has done. For every possible subset of items that you can possibly bring, she has written down the total weight of these items. Alas, she did not write down what items were part of this total, so you do not know what items contributed to each total weight. If the original weights of the items formed a collection $(a_1, \ldots, a_n)$ of non-negative integers, then your friend has written down the multiset

$$S\big((a_1, \ldots, a_n)\big) := \left\{ \sum_{i \in I} a_i \;\middle|\; I \subseteq \{1, \ldots, n\} \right\}.$$

For example, if your friend had two items, and the weights of those two items are $2, 3$, then your friend has written down

- 0, corresponding to the empty set $\{\}$;
- 2, corresponding to the subset $\{2\}$;
- 3, corresponding to the subset $\{3\}$;
- 5, corresponding to the subset $\{2, 3\}$.

You want to reconstruct the weights of all the individual items so you can start using your Knapsack algorithm. It might have happened that your friend made a mistake in adding all these weights, so it might happen that her list is not consistent.

Input

- One line containing a single integer $1 \leq n \leq 18$ the number of items.
- 2^n lines each containing a single integer $0 \leq w \leq 2^{28}$, the combined weight of a subset of the items. Every subset occurs exactly once.

Output

Output non-negative integers $a_1, \ldots, a_n$ on n lines in non-decreasing order such that $S\big((a_1, \ldots, a_n)\big) = \{b_1, \ldots, b_{2^n}\}$, provided that such integers exist. Otherwise, output a single line containing `impossible`.

Example

Input	Output
1 0 5	5
3 7 5 2 4 1 6 3 0	1 2 4
2 0 1 2 4	impossible
2 0 1 1 2	1 1

Problem L. Loo Rolls

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

Your friend Nick needs your help with a hard problem that he came across in real life. Nick has a loo roll of length ℓ centimetres in his bathroom. Every time he visits the toilet, he uses exactly n centimetres of loo roll. When the roll runs out, Nick always goes to the store and buys a new one of length ℓ directly afterwards. However, sometimes the roll runs out even though Nick still needs a non-zero amount of paper. Let us call such an event a *crisis*.

Karen Arnold from Pixabay.

Nick has a clever way of preventing crises from happening: he uses a backup roll. The backup roll is another roll of length ℓ that is hidden somewhere in the bathroom, and when the regular roll runs out even though Nick still needs more paper, he will take that amount from the backup roll. Then he will replace the regular roll directly after the visit.

As you can imagine, this makes crises much less frequent. But still, the backup roll will also slowly run out, and eventually a crisis might still happen. So to generalize this, Nick wants to use several layers of backup rolls. First he will take paper from roll number 1 (the regular roll), if it runs out he will take from roll number 2, then if roll 2 runs out from roll number 3, and so on all the way up to roll number k. After each visit, all the rolls that have run out will be replaced. Nick managed to prove that if he picks a large enough number k, he can actually make it so that crises never happen! Your task is to find the smallest such number k.

Input

The input consists of a single line containing the two integers ℓ and n ($1 \leq n \leq \ell \leq 10^{10}$).

Output

Output the smallest integer k such that crises will never happen when using k layers of rolls (including the regular roll).

Example

Input	Output
`31 6`	`4`
`10000000000 17`	`3`