Problem A. Angry Teacher

Input file:	`angry.in`
Output file:	`angry.out`
Time limit:	2 seconds
Memory limit:	64 megabytes

Mr. O'Cruel is teaching Math to ninth grade students. Students of course are very lazy, so they do not like to do their homework. On the other side, Mr. O'Cruel doesn't like lazy students.

Recently Andrew failed to do his homework again, so he was given a special task. If he doesn't do it, he will be expelled from his school. The task seems very easy, but it is very technical, so it would take a lot of time. Andrew is given a polynomial $p(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ with integer coefficients. He must calculate the value of the polynomial for k successive integer numbers starting from l. Of course writing all these numbers would require too much paper. So as a proof of completing the task, for each number x from l to $l+k-1$ Andrew is asked to provide the sum of squares of m last digits in decimal notation of $p(x)$.

Since Andrew is lazy, he doesn't want to do the task by himself. So he asks you to write the program that calculates the values requested.

Input

The first line of the input file contains n, l, k, and m ($0 \le n \le 10$, $0 \le l \le 10^{1000}$, $1 \le k \le 1\,000$, $1 \le m \le 1\,000$).

The following $n+1$ lines contain coefficients of the polynomial: $a_n, a_{n-1}, \ldots, a_1, a_0$ ($0 \le a_i \le 10^{1000}$).

Output

Output k lines — for x from l to $l+k-1$ output the sum of squares of last m digits of $p(x)$.

Example

angry.in	angry.out

```
3 0 10 2
1
0
2
1
```
	```
	1
	16
	10
	25
	58
	45
	85
	89
	85
	80
	```

Problem B. Beth Tableaux

Input file:	`beth.in`
Output file:	`beth.out`
Time limit:	2 seconds
Memory limit:	64 megabytes

Your program has to check the validity of a given logical formula. If the formula E is invalid (i.e. evaluates to $\mathbf{0}$ for some assignment of boolean values to the variables), you'll also have to find such an assignment of values.

Usually such tasks are solved just by processing all possible assignments of values to the variables occurring in E; if there are k such variables, this requires considering 2^k such assignments. Of course, this is almost impossible for large values of k. Besides, this process is much different from the way people usually check (i.e. prove) logical formulae.

In this task you are to implement a natural way of checking validity of logical formulae based on the so-called *Beth tableaux*. This way of checking validity of formulae models to a certain extent the process of finding proofs or contrary instances used by humans, at least, a short rigorous proof can be usually obtained quite immediately from constructed Beth tableaux.

Let us fix the syntax of logical formulae considered:

- Constants $\mathbf{1}$ and $\mathbf{0}$;
- Variables — letters from A to Z and from a to z;
- Parentheses — if E is a formula, then (E) is another;
- Negation — $\neg E$ is a formula for any formula E;
- Conjunction — $E_1 \wedge E_2 \wedge \cdots \wedge E_n$. Note that the conjunction is evaluated from left to right: $E_1 \wedge E_2 \wedge E_3 = (E_1 \wedge E_2) \wedge E_3$.
- Disjunction — $E_1 \vee E_2 \vee \cdots \vee E_n$. The same remark applies.
- Implication — $E_1 \Rightarrow E_2$. Unlike the previous two operations it is evaluated from right to left: $E_1 \Rightarrow E_2 \Rightarrow E_3$ means $E_1 \Rightarrow (E_2 \Rightarrow E_3)$.
- Equivalence — $E_1 \equiv E_2 \equiv \cdots \equiv E_n$. This expression is by definition computed as follows: $(E_1 \equiv E_2) \wedge (E_2 \equiv E_3) \wedge \cdots \wedge (E_{n-1} \equiv E_n)$.

The operations are listed from the highest priority to the lowest.

Let us define the *principal operation* of a compound formula E as the operation in E which is performed last (of course, formulae consisting just of one constant or variable have no principal operation). Also let us suppose that our formula does not contain equivalence operations since we can always replace $E_1 \equiv E_2$ with $(E_1 \Rightarrow E_2) \wedge (E_2 \Rightarrow E_1)$.

Now we define a *Beth tableau* to be a table with two columns with some formulae written in them. The order formulae are written in the columns and the number of times a formula appears in a column is irrelevant: we can assume for example that all formulae in a column are different and that they are ordered lexicographically.

Informally the left column corresponds to formulae which we want to be true and the right column to those which we want to be false; the whole process can be considered then as an attempt to produce a contrary instance to given formula E. A Beth tableau is said to be *contradictory* if some formula E_* occurs in both columns of this tableau, or if $\mathbf{0}$ occurs in the left column, or if $\mathbf{1}$ occurs in the right column.

At every step of the process we have a collection of Beth tableaux. We start with a single Beth tableau with empty left column and the right column containing just the given formula E. A step of the process consists in choosing a compound formula C from a non-contradictory Beth tableau from the collection and applying a rule chosen by the principal operation of C and by the column (left or right) in which C is situated. As a result some formulae are added to the left or right columns of this Beth tableau (or both); of course, if these formulae that we want to add are already there the Beth tableau won't change, so in this case we say that the corresponding rule is *inapplicable*. There are also some rules that make two copies of the original Beth tableau and add some formulae into one of these copies and some other into the second one. In this case we say that the rule is inapplicable if both Beth tableaux thus obtained were already in the collection before applying the rule.

The process stops when no rule is applicable. Note that all formulae occurring in all Beth tableaux constructed will be sub-formulae of the original formula, so there are only finitely many Beth tableaux that can be obtained and thus only finitely many collections of Beth tableaux can be obtained. Therefore the process will have to stop at some point. If at this point all Beth tableaux in the collection are contradictory, the original formula E has been proved to be valid. Otherwise if τ is a non-contradictory Beth tableau from the collection, any variable x occurring in E must occur in exactly one column of τ (it cannot occur in both columns since τ is not contradictory; it must occur in at least one column since otherwise a rule would be applicable to the shortest formula in τ containing x). Then let us assign **1** to all the variables from the left column and **0** to the variables from the right column of τ. This would give us a contrary instance to E, i.e. an assignment of values to variables for which our original formula E evaluates to **0**.

Now let us list all possible rules. It has been already explained that a rule is defined by the principal operation $*$ of a formula C and the column (left or right) in which C was found. We shall label these rules by $*_l$ and $*_r$ where $*$ is the operation (one of $\wedge$, $\vee$, $\Rightarrow$ or $\neg$). If $*$ is binary, we shall assume that $C = A * B$; otherwise we assume $C = \neg A$.

Let us list the rules that do not increase the number of tableaux (i.e. they just add some formulae to the columns of selected Beth tableau):

$$\wedge_\ell : \begin{array}{c|c} A \wedge B & \cdots \\ \cdots & \\ (A) & \\ (B) & \end{array} \qquad \vee_r : \begin{array}{c|c} \cdots & A \vee B \\ & \cdots \\ & (A) \\ & (B) \end{array} \qquad \Rightarrow_r : \begin{array}{c|c} \cdots & A \Rightarrow B \\ (A) & \cdots \\ & (B) \end{array}$$

$$\neg_\ell : \begin{array}{c|c} \neg A & \cdots \\ \cdots & (A) \end{array} \qquad \neg_r : \begin{array}{c|c} \cdots & \neg A \\ (A) & \cdots \end{array}$$

This means the following: for example, if we apply the rule $\Rightarrow_r$ to the formula $A \Rightarrow B$ in the right column of a Beth tableau, then we have to add formula A to the left column and B to the right column of this tableau.

The next rules produce two Beth tableaux from one. A similar notation is used, but two Beth tableaux are shown for each rule:

$$\wedge_r : \begin{array}{c|c} \cdots & A \wedge B \\ & \cdots \\ & (A) \end{array} \quad \begin{array}{c|c} \cdots & A \wedge B \\ & \cdots \\ & (B) \end{array} \qquad \vee_l : \begin{array}{c|c} A \vee B & \cdots \\ \cdots & \\ (A) & \end{array} \quad \begin{array}{c|c} A \vee B & \cdots \\ \cdots & \\ (B) & \end{array}$$

$$\Rightarrow_l : \begin{array}{c|c} A \Rightarrow B & \cdots \\ \cdots & (A) \\ & \end{array} \quad \begin{array}{c|c} A \Rightarrow B & \cdots \\ \cdots & \\ (B) & \end{array}$$

These three rules are interpreted as follows: if we apply for example $\vee_l$ to the formula $A \vee B$ in the left column of a Beth tableau τ, we have to replace τ with two copies of itself τ' and τ'', and then add A to the left column of τ' and B to the left column of τ''.

Input

The only line of input contains the formula represented as a string consisting of tokens '`0`', '`1`', '`A`'. . . '`Z`', '`a`'. . . '`z`', '`(`', '`)`', '`~`', '`&`', '`|`', '`=>`', '`=`'. The last five tokens stand for $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\equiv$ respectively. Tokens can be separated by an arbitrary number of spaces. The line will contain at most 1 000 characters. The formula in the file will be syntactically correct.

Output

The output file must contain exactly one line. Output just "`true`" if the formula is valid or "`false`" followed by an assignment of variables that invalidates the formula. List the variables in lexicographical order. Adhere to the sample output as strictly as possible.

Example

beth.in	beth.out
`0`	`false`
`1`	`true`
`A`	`false: A=0`
`A|B => A&B`	`false: A=1, B=0`
`A&B => A|B`	`true`
`r=>Y`	`false: Y=0, r=1`
`R=r`	`false: R=0, r=1`
`(A=>B)&(B=>C)=>(A=>C)`	`true`
`(A=>B)=>(~B=>~A)`	`true`
`(A=>B)=>(~A=>~B)`	`false: A=0, B=1`
`(A=a)&(B=b)=>(A&B=a&b)`	`true`
`K|~i|t|t|~e|~n`	`false: K=0, e=1, i=1, n=1, t=0`

Problem C. Collecting Bugs

Input file:	`collect.in`
Output file:	`collect.out`
Time limit:	2 seconds
Memory limit:	64 megabytes

Ivan is fond of collecting. Unlike other people who collect post stamps, coins or other material stuff, he collects software bugs. When Ivan gets a new program, he classifies all possible bugs into n categories. Each day he discovers exactly one bug in the program and adds information about it and its category into a spreadsheet. When he finds bugs in all bug categories, he calls the program *disgusting*, publishes this spreadsheet on his home page, and forgets completely about the program.

Two companies, *Macrosoft* and *Microhard* are in tight competition. *Microhard* wants to decrease sales of one *Macrosoft* program. They hire Ivan to prove that the program in question is disgusting. However, Ivan has a complicated problem. This new program has s subcomponents, and finding bugs of all types in each subcomponent would take too long before the target could be reached. So Ivan and *Microhard* agreed to use a simpler criteria — Ivan should find at least one bug in each subsystem and at least one bug of each category.

Macrosoft knows about these plans and it wants to estimate the time that is required for Ivan to call its program disgusting. It's important because the company releases a new version soon, so it can correct its plans and release it quicker. Nobody would be interested in Ivan's opinion about the reliability of the obsolete version.

A bug found in the program can be of any category with equal probability. Similarly, the bug can be found in any given subsystem with equal probability. Any particular bug cannot belong to two different categories or happen simultaneously in two different subsystems. The number of bugs in the program is almost infinite, so the probability of finding a new bug of some category in some subsystem does not reduce after finding any number of bugs of that category in that subsystem.

Find an average time (in days of Ivan's work) required to name the program disgusting.

Input

Input file contains two integer numbers, n and s ($0 < n, s \leq 1\,000$).

Output

Output the expectation of the Ivan's working days needed to call the program disgusting, accurate to 4 digits after the decimal point.

Example

collect.in	collect.out
`1 2`	`3.000000`

Problem D. Drawing Windows

Input file:	`drawing.in`
Output file:	`drawing.out`
Time limit:	2 seconds
Memory limit:	64 megabytes

Andrew writes portable mailer *KittenMail* for FTN technology networks. This mailer uses text-mode windowed interface like well-known Norton-style shells do.

Now Andrew wants to write a version of his mailer for the new famous operating system *Mycrowslowed Widows Not-Tested 0.4*. It's easy because there is no so much system-dependent code in the mailer. But the windowing subsystem is based on the module providing system-independent interface to the screen buffer functions.

Andrew wrote the trivial code for *Mycrowslowed Widows* displaying the contents of the window buffer on the screen. It was easy because the functions are the same as in other operating systems. But he was so surprised when he ran the mailer! Clearing the screen took about a second! Oops. . . What's wrong?

Andrew started to investigate this problem. Few minutes later, he discovered that any *Mycrowslowed Widows* system call accessing the screen buffer works a huge amount of time — 1/6000 second or more. It's so awful! What can he do?

After hours of thinking, Andrew decided to make some improvements to his code. Now he wants to use *Widows*-specific function that draws a rectangle of characters instead of subsequent calls that display only one character. They perfectly work under other operating systems but are very slow under *Mycrowslowed Widows*. The evident task has arisen. The procedure which redraws window must display its visible parts using the minimal possible number of non-overlapping operations.

It is your task to write the corresponding code for one visible part of the window. Given the part of the rectangular window, write a program which determines the minimal number of non-overlapping rectangles covering this part and finds the optimal way of covering.

Input

The visible part of the window is given by its edge containing only horizontal and vertical segments with integer vertex coordinates. The part does not contain holes and its edge does not intersect or touch itself.

Each segment has length of at least one character.

The first line of the input contains the number n of vertices on the edge of the visible part. The next n lines contain coordinates of vertices in counter-clockwise order (y coordinates grow downwards on the screen).

Horizontal and vertical segments alternate in the input. The last segment is drawn between the last and the first vertices. n does not exceed 400 and absolute values of coordinates are limited to 200.

Output

In the first line output m — the minimal number of rectangular regions covering the visible part of the window.

The next m lines must contain one rectangle description each. The rectangle is specified by four numbers: minimal x, minimal y, maximal x and maximal y.

If there are several optimal solutions, output any of them.

Example

drawing.in	drawing.out

4
0 0
0 1
1 1
1 0
	1
	0 0 1 1

8
0 0
0 1
1 1
1 2
2 2
2 -1
1 -1
1 0
	2
	0 0 1 1
	1 -1 2 2

12
0 0
0 1
1 1
1 2
2 2
2 1
3 1
3 0
2 0
2 -1
1 -1
1 0
	3
	0 0 1 1
	1 -1 2 2
	2 0 3 1

12
0 0
0 3
1 3
1 2
3 2
3 3
4 3
4 0
3 0
3 1
1 1
1 0
	3
	0 0 1 3
	1 1 3 2
	3 0 4 3

Problem E. Ellipse

Input file:	`ellipse.in`
Output file:	`ellipse.out`
Time limit:	2 seconds
Memory limit:	64 megabytes

Alex has got a tedious homework from his geometry teacher as a punishment for his conduct at geometry lessons — Alex didn't do anything while the rest of his class was computing areas of different geometric figures!

Now, Alex has to compute the areas of several ellipses drawn on a sheet of paper torn from a textbook. This paper has a rectangular grid drawn on it which can be used to determine the coordinates of different points. However, the task of finding the area of an ellipse can be quite complicated even in this case, especially if the axes of the ellipse are not vertical or horizontal.

Of course, Alex is very lazy, so he wants you to write a program that would determine the area of an ellipse from the coordinates of five different points lying on it. He would then enter the coordinates of these points for each ellipse himself and thus compute the areas of all ellipses.

Input

The first line of the input contains the number of ellipses k ($1 \leq k \leq 1\,000$). Each of the next k lines contains the coordinates of five points that lie on corresponding ellipse. All coordinates are integer and do not exceed $1\,000$ by their absolute values.

Output

On each of k lines of the output write either "`IMPOSSIBLE`" if the area cannot be determined (e.g. there is no ellipse passing through five given points, or there is more than one such ellipse) or the area itself precise to six digits after decimal point. Note that whenever such an ellipse exists, it always fits completely into the textbook page, i.e. all points (x, y) of the ellipse satisfy inequalities $|x|, |y| \leq 1\,000$.

Example

ellipse.in	ellipse.out

`3`
`5 0  0 5  4 3  3 4  -4 -3`
`6 1  3 2  -2 -3  -3 -2  1 6`
`7 -3  2 7  6 3  5 5  -2 -9`
	`78.539816`
	`IMPOSSIBLE`
	`157.079633`

Problem F. Fool's Day

Input file:	`foolsday.in`
Output file:	`foolsday.out`
Time limit:	2 seconds
Memory limit:	64 megabytes

In a rectangular 2D world everything is rectangular. Rectangular people live in rectangular houses, drive rectangular cars and work in rectangular offices. The only day of a year when everything is not rectangular is April 1st — the Fool's Day. On that day people send each other postcards with jokes and these postcards have a form of parallelograms. To support this tradition the rectangular world industry produces different parallelogram postcards and envelopes.

Mr. Rect Squarowski wants to start a new business. He would like to provide a web service that determines whether a Fool's Day postcard fits Fool's Day envelope. Help Mr. Squarowski to earn his next billion of money rects.

Input

The first line of the input file contains integer number n ($1 \le n \le 1\,000$) — the number of test cases. Each test case consists of six integer numbers. The first three numbers describe an envelope, the last three — a postcard.

Three numbers describing each parallelogram are the lengths of its adjacent sides and its diagonal respectively. Specified lengths will not be greater than 1 000.

It is guaranteed that the corresponding parallelogram exists and it is not degenerate.

Output

For each test case output a single line. Print "`Yes`" if a postcard fits an envelope or "`No`" if it doesn't.

Example

foolsday.in	foolsday.out

3
3 4 6 4 3 6
6 6 7 5 5 8
11 11 11 10 10 2
	Yes
	Yes
	No

Problem G. Graveyard Design

Input file:	`graveyard.in`
Output file:	`graveyard.out`
Time limit:	2 seconds
Memory limit:	64 megabytes

King George has recently decided that he would like to have a new design for the royal graveyard. The graveyard must consist of several sections, each of which must be a square of graves. All sections must have different number of graves.

After a consultation with his astrologer, King George decided that the lengths of section sides must be a sequence of successive positive integer numbers. A section with side length s contains s^2 graves.

George has estimated the total number of graves that will be located on the graveyard and now wants to know all possible graveyard designs satisfying the condition. You were asked to find them.

Input

Input file contains n — the number of graves to be located in the graveyard ($1 \le n \le 10^{14}$).

Output

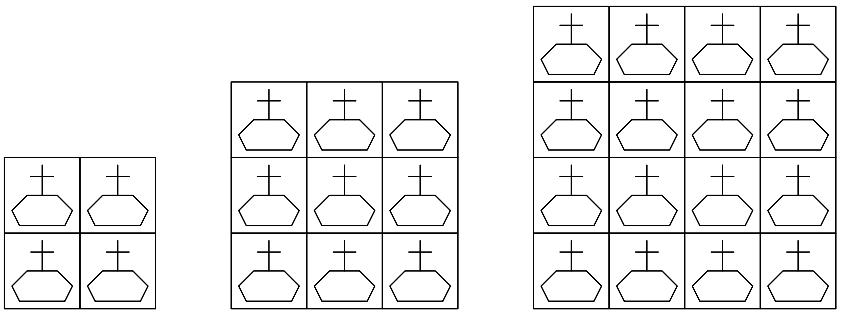
On the first line of the output file print k — the number of possible graveyard designs. Next k lines must contain the descriptions of the graveyards. Each line must start with l — the number of sections in the corresponding graveyard, followed by l integers — the lengths of section sides (successive positive integer numbers).

Example

`graveyard.in`	`graveyard.out`
29	1 3 2 3 4

2030	2
	4 21 22 23 24
	3 25 26 27

The picture below illustrates the graveyard for the first example.



Problem H. Honey and Milk Land

Input file:	`honey.in`
Output file:	`honey.out`
Time limit:	2 seconds
Memory limit:	64 megabytes

Bad rumors are spreading over the *Land of Honey and Milk*. Informed people say that the milk in the famous grid of milk rivers is turning sour. Of course, the security service quickly found out that the people are informed by the *Kingdom of Tar*, which is jealous to tourist popularity of the land. However, this discovery does not help to stop these rumors. The government wants to prevent crisis of the tourist industry, so it wants to establish daily monitoring of the rivers.

A new *Milk Security Department* is established, which is responsible for preventing the milk from turning sour. It's equipped with powerful boilers and pasteurizer, so any danger for the milk can be quickly neutralized. To better fight the new threat, the department needs to know about possible dangers beforehand. They have a helicopter, capable to check milk freshness. The equipment is perfect. It's enough just to cross a river in any place in order to detect all its potentially dangerous places.

To start the *Milk Security Department* operations, the government needs to add funding of the Service to the Land budget. One of the issues is the morning route of the helicopter. The helicopter should check all the rivers in the shortest time. They need to determine the price of this flight to add it to the budget.

The grid consists of two sets of milk rivers. Rivers from the first set run from North to South, rivers from the second set — from East to West. The rivers are straight. The rivers from each set are parallel and the distance between the adjacent rivers is known. There are n rivers, running from North to South and e rivers, running from East to West.

The government needs to determine the minimal morning flight cost. Each kilometer costs 1 honey barrel, the Land national currency. The cost of take-off and landing is not included into this cost. You may freely choose the starting and ending points of the flight.

Input

The first line of the input file contains n and e ($1 \le n, e \le 1\,000$). The second line contains $n-1$ integer numbers that represent distances (in kilometers) between adjacent rivers running from North to South, listed from East to West. The third line contains $e-1$ integer numbers that represent distances (also in kilometers) between adjacent rivers running from East to West, listed from North to South. The distance between any two adjacent rivers does not exceed 27 kilometers.

Output

Output the minimal morning flight cost in honey barrels. Since there is no smaller denomination, you must output the minimal integer number of honey barrels that would be sufficient to support the flight.

Example

`honey.in`	`honey.out`

`2 1`
`1`
	`1`

`10 10`
`2 2 2 2 2 2 2 2 2`
`2 2 2 2 2 2 2 2 2`
	`26`

`1 1`	`0`

Problem I. Incredible! Impossible!

Input file:	`incredible.in`
Output file:	`incredible.out`
Time limit:	2 seconds
Memory limit:	64 megabytes

“Give me the sums of each column and of each row of the rectangular table $n \times m$”, — Alex said, — “and I will tell you whether such a table exists, and if exists for a small fee I can create an example of such table”. “Incredible! Impossible!”, — say all his classmates, — “So many numbers! You must be a true genius!”.

But Basil does not like that Alex becomes the most known person at the school.

— “I say! I am Basil the greatest! Given a rectangular table $n \times m$ and the sums of rows and columns I will tell you a number of possible tables with non-negative integers satisfying these conditions”.

— “You boast! I bet five dollars you can’t do it even for $n \times 3$”, — Alex says.

— “I bet five that I can!”, — says Basil.

Tomorrow is the contest. Alex will create some tables with size $n \times 3$, and tell Basil the sums and the dimensions. All boys and girls do the stakes who will be the winner!

You are the friend of Alex. He wants to create some hard data sets for Basil, and he needs to have a method to calculate the answer. Because Alex can’t solve such kind of tasks, he asked you to write a program that will do it for him.

Alex needs only last seventeen digits to check the answers. So you must calculate the number of possible tables taken modulo 10^{17}.

Input

The first line contains four numbers: n, c_1, c_2, c_3, where n is the number of rows, c_i are the sums of columns. n numbers follow, each is the sum of the corresponding row. n and all sums are non-negative integers. They are not greater than 125.

Output

On the first line output the number of possible tables taken modulo 10^{17}.

Example

`incredible.in`	`incredible.out`

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

`3 1 1 1`
`1 1 1`
	`6`

`2 1 1 1`
`1`
`2`
	`3`

Problem J. Jackpot

Input file:	`jackpot.in`
Output file:	`jackpot.out`
Time limit:	2 seconds
Memory limit:	64 megabytes

The *Great Dodgers* company has recently developed a brand-new playing machine.

You put a coin into the machine and pull the handle. After that it chooses some integer number. If the chosen number is zero you win a jackpot. In the other case the machine tries to divide the chosen number by the lucky numbers $p_1, p_2, \ldots, p_n$. If at least one of the remainders is zero — you win.

Great Dodgers want to calculate the probability of winning on their machine. They tried to do it, but failed. So *Great Dodgers* hired you to write a program that calculates the corresponding probability.

Unfortunately, probability theory does not allow you to assume that all integer numbers have equal probability. But one mathematician hinted you that the required probability can be approximated as the following limit:

$$\lim_{k \to \infty} \frac{S_k}{2k+1}.$$

Here S_k is the number of integers between $-k$ and k that are divisible by at least one of the lucky numbers.

Input

Input file contains n — the number of lucky numbers ($1 \le n \le 16$), followed by n lucky numbers ($1 \le p_i \le 10^9$).

Output

It is clear that the requested probability is rational. Output it as an irreducible fraction.

On the first line of the output file print the numerator of the winning probability. On the second line print its denominator. Both numerator and denominator must be printed without leading zeroes. Remember that the fraction must be irreducible.

Example

jackpot.in	jackpot.out
1 2	1 2
2 4 6	1 3

Problem K. K-th Number

Input file:	`kth.in`
Output file:	`kth.out`
Time limit:	2 seconds
Memory limit:	64 megabytes

You are working for *Macrohard* company in data structures department. After failing your previous task about key insertion you were asked to write a new data structure that would be able to return quickly k-th order statistics in the array segment.

That is, given an array $a[1 \ldots n]$ of different integer numbers, your program must answer a series of questions $Q(i, j, k)$ in the form: "*What would be the k-th number in $a[i \ldots j]$ segment, if this segment was sorted?*"

For example, consider the array $a = (1, 5, 2, 6, 3, 7, 4)$. Let the question be $Q(2, 5, 3)$. The segment $a[2 \ldots 5]$ is $(5, 2, 6, 3)$. If we sort this segment, we get $(2, 3, 5, 6)$, the third number is 5, and therefore the answer to the question is 5.

Input

The first line of the input file contains n — the size of the array, and m — the number of questions to answer ($1 \leq n \leq 100\,000$, $1 \leq m \leq 5\,000$).

The second line contains n different integer numbers not exceeding 10^9 by their absolute values — the array for which the answers should be given.

The following m lines contain question descriptions, each description consists of three numbers: i, j, and k ($1 \leq i \leq j \leq n$, $1 \leq k \leq j - i + 1$) and represents the question $Q(i, j, k)$.

Output

For each question output the answer to it — the k-th number in sorted $a[i \ldots j]$ segment.

Example

kth.in	kth.out

7 3
1 5 2 6 3 7 4
2 5 3
4 4 1
1 7 3
	5
	6
	3