Data Centers

Problem Name	DataCenters
Input File	standard input
Output File	standard output
Time limit	2 seconds
Memory limit	256 megabytes

GoncaSoft is an internet company that runs many services and has n data centers worldwide. Each data center has a number of available machines. For security and redundancy reasons, one or more copies of each service are running at the same time. Each copy runs in a separate data center, and requires a number of machines to run on. All the copies of a given service require the same number of machines.

When GoncaSoft plans to launch a new service i that requires c_i copies, each running on m_i machines, it sorts the data centers in descending order by their current available machines, and then uses m_i machines in each of the top c_i data centers.

Please calculate the remaining available machines in the data centers after launching s services in a given order.

Input

The first line of the input contains two space-separated integers n and s, representing the number of data centers GoncaSoft has and the number of new services GoncaSoft wants to launch.

The next line contains n space-separated integers, representing the number of available machines in each of the n data centers, before any services are launched.

The next s lines describe the services that will be launched: the i^{th} line contains two space-separated integers m_i and c_i, representing the number of machines and the number of copies the i^{th} service requires.

Output

Output one line containing n space-separated integers sorted in **descending order**, representing the number of remaining available machines in the data centers after all services have launched.

Constraints

- $1 \leq n \leq 100\,000$ and $0 \leq s \leq 5\,000$.
- Each data center has at most 10^9 machines initially.
- $1 \leq m_i \leq 10^9$ for each service i such that $1 \leq i \leq s$.
- $1 \leq c_i \leq n$, for each service i such that $1 \leq i \leq s$.
- The data centers will always have enough machines for the new services.

Scoring

- Subtask 1 (12 points): $n \leq 100$, $s = 0$.
- Subtask 2 (12 points): $n \leq 100$, $s \leq 10$.
- Subtask 3 (9 points): $n \leq 50\,000$, $s \leq 100$.
- Subtask 4 (26 points): Each data center has initially at most $1\,000$ machines.
- Subtask 5 (18 points): $c_i = 1$ for all services from 1 to s.
- Subtask 6 (23 points): No further constraints.

Example

standard input	standard output
5 4 20 12 10 15 18 3 4 4 1 1 3 4 2	11 10 10 9 8

Explanation

Step	Available Machines	Operations
Beginning	20 12 10 15 18	
Service #1: before launching	20 18 15 12 10	Sort the data centers in descending order.
Service #1: after launching	17 15 12 9 10	Use 3 machines in each of the top 4 data centers.
Service #2: before launching	17 15 12 10 9	Sort the data centers in descending order.
Service #2: after launching	13 15 12 10 9	Use 4 machines in the top data center.
Service #3: before launching	15 13 12 10 9	Sort the data centers in descending order.
Service #3: after launching	14 12 11 10 9	Use 1 machine in each of the top 3 data centers.
Service #4: before launching	14 12 11 10 9	Sort the data centers in descending order.

Service #4: after launching	10 8 11 10 9	Use 4 machines in each of the top 2 data centers.
End	11 10 10 9 8	Sort the data centers in descending order.

Superpiece

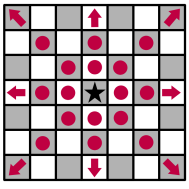
Problem Name	Superpiece
Input File	standard input
Output File	standard output
Time limit	1 second
Memory limit	256 megabytes

You are given an infinite chessboard. In this task, a chessboard is an infinite two-dimensional grid of squares, where each square is indexed by a pair of integers (r, c), denoting the row and the column respectively. The only piece currently present on the chessboard is the **superpiece**. You are given a list of valid moves of your superpiece, which will be specified as a non-empty string containing a subset of the characters in "QRBNKP". In each turn, the superpiece can move as one of the given chess pieces. The superpiece is initially positioned at square (a, b). Calculate the minimum number of moves needed to reach the square (c, d).

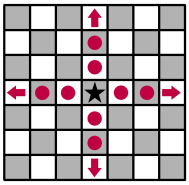
The subset of the chess rules applicable for this problem are given below.

There are six types of pieces: queen, rook, bishop, knight, king and pawn. They move the following way:

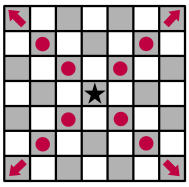
- The **Queen** (denoted by **'Q'**) can move to any square in the same row, column or diagonal as the square it is currently in. Formally, for any integer $k \neq 0$, a queen can move from (a, b) to $(a, b + k)$, $(a + k, b)$, $(a + k, b + k)$ and $(a + k, b - k)$.



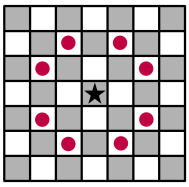
- The **Rook** (denoted by **'R'**) can move to any square in the same row or in the same column as the square it is currently in. Formally, for any integer $k \neq 0$, a rook can move from (a, b) to $(a + k, b)$ and $(a, b + k)$.



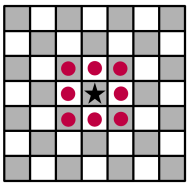
- The **Bishop** (denoted by **'B'**) can move to any square in the same diagonal as the square it is currently in. Formally, for any integer $k \neq 0$, a bishop can move from (a, b) to $(a + k, b + k)$, and $(a + k, b - k)$.



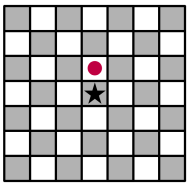
- The **kNight** (denoted by **'N'**) can move in the shape of an 'L': that is, it first moves two squares in a particular direction followed by a move of one sqaure in an perpendicular direction. Formally, a knight can move from (a, b) to $(a + 1, b + 2)$, $(a + 1, b - 2)$, $(a + 2, b + 1)$, $(a + 2, b - 1)$, $(a - 2, b + 1)$, $(a - 2, b - 1)$ $(a - 1, b + 2)$ and $(a - 1, b - 2)$.



- The **King** (denoted by **'K'**) can move to any of the eight squares directly adjacent to the current square. Formally, a king can move from (a, b) to $(a, b + 1)$, $(a, b - 1)$, $(a + 1, b)$, $(a - 1, b)$, $(a + 1, b + 1)$, $(a + 1, b - 1)$, $(a - 1, b + 1)$ and $(a - 1, b - 1)$.



- The **Pawn** (denoted by **'P'**) can move exactly one square up. Formally, a pawn can move from (a, b) to $(a + 1, b)$.



Note that the other rules or moves that you might know about chess do not apply in this problem; please only use the ones listed above.

Also, note that while the symbol denoting the chess piece is often the first letter of its name in English, it is the second letter for the "kNight" (to avoid confusion with the "King").

Input

The first line of the input contains an integer q, representing the number of queries your program will be tested on. Each two of the following lines describe a query:

- The first line of a query contains a non-empty string specifying the set of chess pieces the superpiece can move as. This string contains a subset of the characters in the uppercase string "QRBNKP", with the contained characters appearing **in the same order**. In other words, it is in the form of a sub-sequence of "QRBNKP".
- The second line of a query contains four space-separated integers a, b, c, d - the original and the target position of the superpiece. It is guaranteed that $(a, b) \neq (c, d)$, that is, the original position is different from the target.

Output

For each of the q queries, output a single line containing an integer m representing the minimum number of moves the superpiece needs to reach the target from its original position for that query. If it is not possible to reach the target from the original position for a query, output -1 instead.

Constraints

- $1 \leq q \leq 1000$
- $-10^8 \leq a, b, c, d \leq 10^8$ for each query.
- The chess board is infinite in all directions.

Scoring

- Subtask 1 (12 points): No 'N' character and a guaranteed 'Q' character in the first line of each query.
- Subtask 2 (9 points): Guaranteed 'Q' and 'N' characters (both) in the first line of each query.
- Subtask 3 (13 points): No 'Q' character and a guaranteed 'R' character in the first line of each query.
- Subtask 4 (8 points): The first line of each query is always "B".
- Subtask 5 (6 points): No 'Q' or 'R' characters and a guaranteed 'B' character in the first line of each query.
- Subtask 6 (31 points): The first line of each query is always "N".
- Subtask 7 (8 points): No 'Q', 'R', or 'B' characters and a guaranteed 'N' character in the first line of each query.
- Subtask 8 (7 points): No 'Q', 'R', 'B', or 'N' characters and a guaranteed 'K' character in the first line of each query.
- Subtask 9 (6 points): The first line of each query is always "P".

Note that subtasks are **not** ordered in the expected order of their difficulty.

Examples

standard input	standard output
2 NKP 3 3 5 1 NKP 2 6 5 3	2 2
2 B 2 8 3 6 B 2 8 5 5	-1 1
2 Q 3 3 4 5 QR 4 1 1 4	2 1

Explanation

Test case 1

In the first query, we are asked to go from $(3, 3)$ to $(5, 1)$, using the moves of knight, king and pawn. There are multiple ways of doing this in exactly 2 moves, for example:

- Move as a pawn to $(4, 3)$, then as a knight to $(5, 1)$.
- Move as a knight to $(5, 2)$, then as a king to $(5, 1)$.
- Move as a king to $(4, 2)$, and then again as a king to $(5, 1)$.

There is no way of achieving this by fewer than two moves - we would need a bishop or a queen to do that.

In the second query, we are asked to go from $(2, 6)$ to $(5, 3)$. Again, the optimal solution is to use two moves. This time, both of these moves have to be knight moves, with the intermediate square being $(4, 5)$ or $(3, 4)$.

Test case 2

In the first query, we are asked to go from $(2, 8)$ to $(3, 6)$. Provided only the bishop's moves, it is not possible to do this.

In the second query, we are asked to go from $(2, 8)$ to $(5, 5)$, again using only bishop's moves. It is possible to do this in one move.

Test case 3

In the first query, we are asked to go from $(3, 3)$ to $(4, 5)$ using the queen's moves. It is possible to do this in two moves, for instance, by using $(4, 4)$ as an intermediate point.

In the second query, we are asked to go from $(4, 1)$ to $(1, 4)$, using the moves of queen and rook. It is possible to do this in one move.

Toy Design

Problem Name	ToyDesign
Input File	Interactive Task
Output File	Interactive Task
Time limit	1 second
Memory limit	256 megabytes

You are working for a company that designs toys. A new toy that is being created works like this: There are n pins, numbered from 1 to n, sticking out of a box. Some pairs of pins are connected with wires inside the box. (In other words, the pins and the wires form an undirected graph, where pins are the vertices and wires are the edges.) The wires are not visible from outside, and the only way to find out something about them is to use a **tester** on the pins: We can pick two pins i and j such that $i \neq j$, and the tester will tell if those two pins are connected inside the box, either directly or indirectly. (Thus, the tester tells whether there is a path between those pins in the graph.)

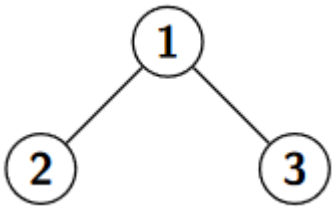
We will call the set of connections inside the box the **design** of the toy.

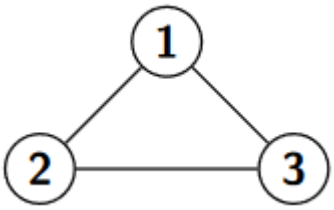
You are using a specialized software to query and create these designs. This software works like this: It starts with some design of the toy which we denote as "design 0". It does not show you the connections inside the box for this design. Instead, you can repeatedly perform the following three-step operation:

1. You pick a design number a and two pin numbers i and j such that $i \neq j$.
2. The software tells you what would happen if we use the tester on those two pins. In other words, it tells you if pins i and j are (directly or indirectly) connected in design a.
3. Also, if the pins were not directly or indirectly connected in design a, then it creates a new design which has all connections from design a plus one additional direct connection between i and j. This design is given the next available design number. (So, the first design created in this way will have number 1, then number 2, and so on.) Note that this does not change design a, just creates a new design that has the additional connection.

Your goal is to learn as much as possible about design 0 by using this operation.

Note that it is not always possible to determine the exact set of connections for design 0, because there is no way to distinguish direct and indirect connections. For example, consider the following two designs with $n = 3$:





The tester would report any pair of pins as connected for both designs, so we will not be able to distinguish them using the software described above.

Your goal is to determine any design that is equivalent to design 0. Two designs are **equivalent** if the tester reports the same result in both designs for all pairs of pins.

Implementation

This is an interactive problem. You must implement a function

```
void ToyDesign(int n, int max_ops);
```

that determines a design that is *equivalent* to design 0. Your implementation should achieve this goal by calling two functions as described below. The first function you can call is:

```
int Connected(int a, int i, int j);
```

where $1 \leq i, j \leq n$, $i \neq j$, $a \geq 0$, and a must not exceed the number of designs created so far. If pins i and j are (directly or indirectly) connected in design a, then it will return a back. Otherwise, it will return the number of designs created so far plus one, which becomes the number assigned to the new design which has all the connections of design a plus the connection between i and j. The function `Connected` can be called at most `max_ops` times.

When your program is done with the `Connected` operations, it should describe a design that is equivalent to design 0. To describe a design, the program should call:

```
void DescribeDesign(std::vector<std::pair<int,int>> result);
```

The parameter `result` is a vector of integer pairs describing the direct connections between the pins. Each pair corresponds to one connection and should contain the two pin numbers of the connection. There must be at most one direct connection between each (unordered) pair of pins, and no direct connections between a pin and itself. Calling this function terminates the execution of your program.

Constraints

- $2 \leq n \leq 200$

Scoring

- Subtask 1 (10 points): $n \leq 200$, *max_ops* $= 20\,000$
- Subtask 2 (20 points): $n \leq 8$, *max_ops* $= 20$
- Subtask 3 (35 points): $n \leq 200$, *max_ops* $= 2\,000$
- Subtask 4 (35 points): $n \leq 200$, *max_ops* $= 1\,350$

Sample Interaction

Contestant action	Grader action	Explanation
	`ToyDesign(4, 20)`	There are 4 pins in the toy. You need to determine any design that is equivalent to design 0 by calling `Connected` at most 20 times.
`Connected(0, 1, 2)`	`Returns 1.`	Pins 1 and 2 are not connected directly or indirectly in design 0. New design 1 is created.
`Connected(1, 3, 2)`	`Returns 2.`	Pins 3 and 2 are not connected directly or indirectly in design 1. New design 2 is created.
`Connected(0, 3, 4)`	`Returns 0.`	Pins 3 and 4 are connected directly or indirectly in design 0. No new design is created.
`DescribeDesign({{3, 4}})`	-	We describe a design that has only one connection: Pins 3 and 4.

Sample Grader

The provided sample grader, `grader.cpp`, in the task attachment `ToyDesign.zip`, reads the input from the standard input in the following format:

- The first line contains the number of pins, n, the number of direct connections, m and *max_ops*

- The following m lines contain direct connections as tuples of pins.

The sample grader reads the input and calls the `ToyDesign` function in user's solution. The grader will output one of the following messages, based on the behaviour of your solution:

- "Wrong answer: Number of operations exceeds the limit.", if the number of calls to `Connected` exceeds *max_ops*
- "Wrong answer: Wrong design id.", if the parameter a to a call to `Connected` is the number of a design that does not exist at the moment the call was made.
- "Wrong answer: Incorrect design.", if the design described via `DescribeDesign` is not equivalent to design 0.
- "OK!" if the design described via `DescribeDesign` is equivalent to design 0.

To compile the sample grader with your solution, you may use the following command at the terminal prompt:

```
g++ -std=gnu++11 -O2 -o solution grader.cpp solution.cpp
```

where `solution.cpp` is your solution file to be submitted to CMS. To run the program with the sample input provided in the attachment, type the following command into the terminal prompt:

```
./solution < input.txt
```

Chika Wants to Cheat

Problem Name	Cheat
Input File	Interactive task
Output File	Interactive task
Time limit	2 seconds
Memory limit	512 megabytes

Chika has a deck of q playing cards numbered with various positive integers. She wants to play some games with her friends from the Shuchi'in Academy student council using these cards, but she also wants to win, so she decides to secretly mark the back of the cards in her deck.

The cards are all square-shaped and of size 2×2, where the bottom-left corner has coordinates $(0, 0)$ and the top-right corner has coordinates $(2, 2)$. Chika draws a certain pattern on the back of each card, so that she will later know, by looking at the pattern, which number is written on the front of the card. She draws such a pattern using the following procedure: As many times as she wants (possibly 0 times), she picks two distinct points A and B that have integer coordinates relative to the bottom-left corner of the card and draws a **straight line segment** between them.

Chika will only draw **valid** segments, that is, segments between two points A and B for which there is no another point C (distinct from A and B) with integer coordinates that also lies on the segment. For example, the segment between $(0, 0)$ and $(2, 2)$ is **not valid** as it contains point $(1, 1)$, but segments between $(0, 0)$ and $(1, 1)$ and between $(1, 1)$ and $(2, 2)$ are both **valid**, and Chika can even draw both of them on the same pattern. Also, note that the segments have no direction: A segment drawn from A to B is **identical** to both itself and the segment drawn in the reverse direction, from B to A.

Importantly, Chika wants to make sure that she will recognize her cards regardless of how they are rotated. A card can be rotated 0, 90, 180 or 270 degrees counter-clockwise with respect to the original orientation.

Your task is to help Chika design the patterns for the q cards in her deck and then recognize those cards later on.

Implementation

This is an interactive task with two stages, ***<u>each stage involving a separate run of your program</u>***. You need to implement two functions:

- A `BuildPattern` function that returns the pattern to be drawn on the back of a given card. This function will be called q times in the first stage.
- A `GetCardNumber` function that returns the number of a (possibly rotated) card that carries a given pattern drawn in the first stage. This function will be called q times in the second stage.

The first function

```
std::vector<std::pair<std::pair<int, int>, std::pair<int, int>>> BuildPattern(int n);
```

takes a single parameter n, the number that is written on the front of the card. You need to return a `std::vector` containing the segments that Chika draws as a pattern on the back of the card to recognize it later. A segment is represented as a `std::pair` of points, and a point is represented as a `std::pair` (x, y) of integer coordinates relative to the bottom-left corner of the card, where $0 \leq x, y \leq 2$. All segments that Chika draws need to be valid and pairwise non-identical. It is guaranteed that all q calls to BuildPattern receive different values for the parameter n.

After receiving all the patterns for the q cards, the grader can do any of the following operations, any number of times, on each of the patterns:

- Rotate the entire pattern by 0, 90, 180 or 270 degrees counter-clockwise.
- Modify the order of segments in the `std::vector` representation of the pattern.
- Change the order of the endpoints of a segment in the pattern. (A segment drawn from A to B may become the identical segment from B to A.)

The second function,

```
int GetCardNumber(std::vector<std::pair<std::pair<int, int>, std::pair<int, int>>> p);
```

takes a single parameter p, a `std::vector` of segments describing the pattern that is drawn by Chika on the back of the card, based on the return value of an earlier call to your `BuildPattern` function. The function must return the number n written on the front of the card. Recall that the pattern p is not necessarily in the original form that is returned by `BuildPattern`; it may have been subject to the three operations mentioned above. It is also possible that the order of the cards is different from the order they were given in the first stage, but it is guaranteed that each card will be used exactly once.

Constraints

- $1 \leq q \leq 10\,000$.
- $1 \leq n \leq 67\,000\,000$ for all calls to function `BuildPattern`.
- Note that there exist algorithms to construct patterns such that one can recognize $67\,000\,000$ different cards.

Scoring

- Subtask 1 (2 points): $n \leq 2$.
- Subtask 2 (9 points): $n \leq 25$.
- Subtask 3 (15 points): $n \leq 1\,000$ and the grader will **not rotate** the patterns between stages 1 and 2. (The grader **may** perform the other two operations.)
- Subtask 4 (3 points): $n \leq 16\,000\,000$ and the grader will **not rotate** the patterns between stages 1 and 2. (The grader **may** perform the other two operations.)
- Subtask 5 (24 points): $n \leq 16\,000\,000$.
- Subtask 6 (18 points): $n \leq 40\,000\,000$.
- Subtask 7 (29 points): No further constraints.

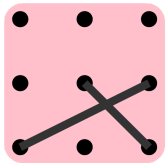
Sample Interaction

Function call	Return value	Explanation
First stage begins.	-	-
`BuildPattern(3)`	`{{{0, 0}, {2, 1}}, {{1, 1}, {2, 0}}}`	We have to create a pattern for number 3 on the 2×2 sized card. We decide to draw 2 segments: - between $(0, 0)$ and $(2, 1)$, - between $(1, 1)$ and $(2, 0)$.
`BuildPattern(1)`	`{{{0, 1}, {0, 0}}}`	We have to create a pattern for number 1 on the 2×2 sized card. We decide to draw 1 segment: - between $(0, 1)$ and $(0, 0)$.
First stage ends.	-	-
Second stage begins.	-	-
`GetCardNumber( {{{0, 0}, {0, 1}}})`	`1`	We get a pattern made up of 1 segment: - between $(0, 0)$ and $(0, 1)$. This is the same pattern as we would get from drawing the segment: - between $(0, 1)$ and $(0, 0)$. which is exactly the same pattern with the same orientation (rotated by 0 degrees) we returned on the second

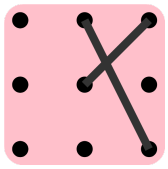
		call to function `BuildPattern`. Therefore, we return 1.
`GetCardNumber({{{1, 1}, {2, 2}}, {{1, 2}, {2, 0}}})`	`3`	We get a pattern made up of 2 segments: - between $(1, 1)$ and $(2, 2)$, - between $(1, 2)$ and $(2, 0)$. This is the pattern we returned on the first call to the `BuildPattern` function, rotated by 90 degrees counter-clockwise. Therefore, we return 3.
Second stage ends.	-	-

The next three pictures represent, in order:

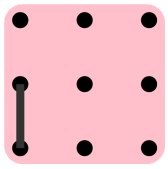
- The pattern returned as output by the first call to `BuildPattern`:



- The pattern received as parameter by the second call to `GetCardNumber`, which is the first pattern after it is rotated by 90 degrees counter-clockwise.



- The pattern returned as output by the second call to `BuildPattern`, which is also the same pattern received as argument by the first call to `GetCardNumber`.



Sample Grader

The provided sample grader, `grader.cpp`, in the task attachment `Cheat.zip`, reads an integer q from standard input and then will do the following steps q times:

- Read an integer n from standard input.
- Call `BuildPattern(n)` and store the return value in a variable p.
- Call `GetCardNumber(p)` and print the return value to standard output.

You can modify your grader locally if you wish to do so.

To compile the sample grader with your solution, you may use the following command at the terminal prompt:

```
g++ -std=gnu++11 -O2 -o solution grader.cpp solution.cpp
```

where `solution.cpp` is your solution file to be submitted to CMS. To run the program with the sample input provided in the attachment, type the following command in the terminal prompt:

```
./solution < input.txt
```

Please note that, unlike the sample grader, the real grader on CMS will perform the first stage and second stage in separate executions of your program.