Softuniada 2019

1. Digitivision

You will be given 3 digits. Your task is to find if there is any 3-digit number:

- **Formed** by the **given digits**
- That is **divisible** (**without remainder**) by the **sum** of the **given digits**

If there is **any number** fulfilling the conditions specified above, you should print "**`Digitivision successful!`**".

If there is **no such** number, you should print "**`No digitivision possible.`**".

Input

The input comes in 3 input lines, each of them containing a single digit.

Output

Depending on whether a "**`digitivision`**" is possible or not you should print one of the following lines:

- "**`Digitivision successful!`**", if there is a successful "**`digitivision`**" without remainder.
- "**`No digitivision possible.`**", if there is no possible "**`digitivision`**" without remainder.

Constraints

- The input lines will contain **only digits** – integers in **range [0, 9]**.
- Allowed time / memory: 100ms / 16MB.

Examples

Input	Output	Comment
6 2 1	`Digitivision successful!`	The **sum** of the **digits** is **9**. We start forming the numbers: **`621 / 9 = 69`** **`612 / 9 = 68`** **`261 / 9 = 29`** **`216 / 9 = 24`** **`162 / 9 = 18`** **`126 / 9 = 14`** There are **6 possible divisions** without **remainder**. We needed only **1**. Hence, the "**`digitivision`**" is **possible**.
3 3 4	`No digitivision possible!`	The **sum** of the **digits** is **`10`**. We start forming the numbers: **`334 / 10 = 33.4 (remainder 0.4)`** **`343 / 10 = 34.3 (remainder 0.3)`** **`433 / 10 = 43.3 (remainder 0.3)`**

		There are **no possible divisions** without **remainder**. Hence, the "**`digitivision`**" is **NOT possible**.

2. Crocs

You have been tasked to draw a **croc shoe**, by an unusually rich client, for an unusually low price.

You will be given **N** – an **odd integer number**. You draw a croc with **width** – **N * 5** and **height** – **N * 4 + 2**. For more details on the form of the croc, you should see the examples below.

Input

The input will consist of a **single line**, on which you will receive an **odd integer number**.

Output

The output should be a correctly-drawn croc shoe, just like the examples below.

Constraints

- The integer **N** will always be an **odd** number in **range [0, 100]**.
- Allowed time / memory: 100ms / 16MB.

Examples

Input	Output	Input	Output	Comment

3	`   #########   `
	`###         ###`
	`### # # # # ###`
	`###  # # #  ###`
	`### # # # # ###`
	`###  # # #  ###`
	`### # # # # ###`
	`###         ###`
	`###############`
	`### # # # # ###`
	`###############`
	`### # # # # ###`
	`###############`
	`   #########   `
		5	`     ###############     `
			`     ###############     `
			`#####               #####`
			`##### # # # # # # # #####`
			`#####  # # # # # #  #####`
			`##### # # # # # # # #####`
			`#####  # # # # # #  #####`
			`##### # # # # # # # #####`
			`#####  # # # # # #  #####`
			`##### # # # # # # # #####`
			`#####  # # # # # #  #####`
			`##### # # # # # # # #####`
			`#####               #####`
			`#########################`
			`##### # # # # # # # #####`
			`#########################`
			`##### # # # # # # # #####`
				The **head lines** (**first** and **last** several lines) are exactly **N / 2** (**integer division**) by count.

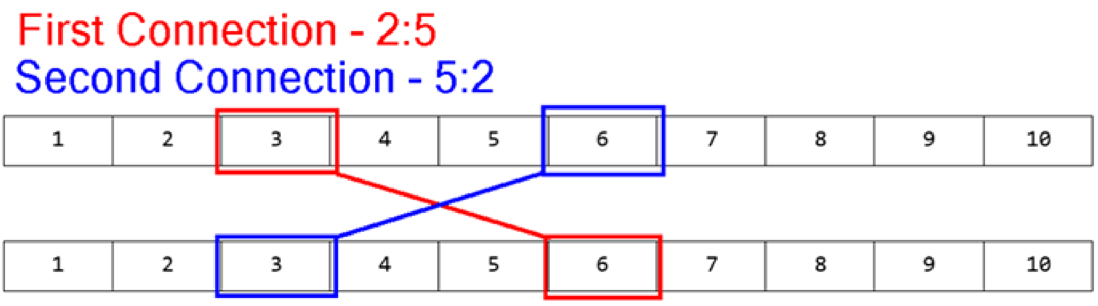
			`#########################`
			`##### # # # # # # # #####`
			`#########################`
			`###############`
			`###############`	

3. Nexus

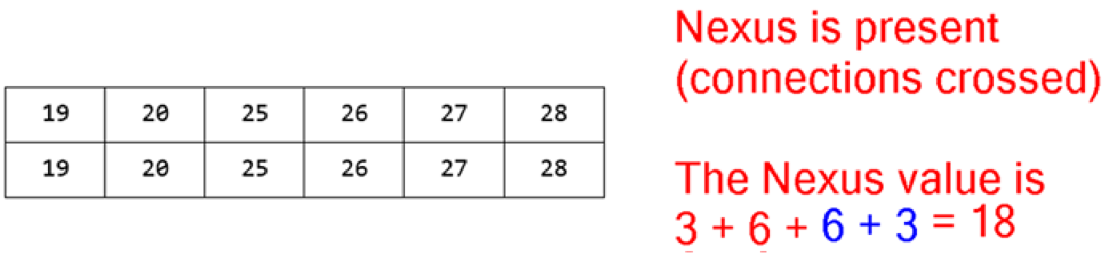
While studying arrays of various atoms, Doctor Sanity manager to discover a strange behaviour – while connecting 2 pairs of atoms from 2 parallel arrays, a nexus is initiated. If the 2 connections cross each other, the nexus disperses and loads all other atoms with nuclear value. He wanted to create an algorithm simulating this behaviour, but he's too lazy… So you'll have to write it for him.

You will receive **2 sequences** of **integers**, separated by **spaces** – the **2 arrays** of atoms.

After that you will start receiving **2 pairs** of **indices** – the **2 connections** in the **arrays**. Each connection will be in the **form** of an **index** from the **first array** and another **index** from the **second array**.



You must check if the 2 connections **cross each other**. If they **do**, you must **remove all elements between** them from **both arrays**, and you must **increase all integers**, that are **left afterwards** in **both arrays**, with the **summed up value** of the **connected elements** (**Nexus value**).



19	20	25	26	27	28
19	20	25	26	27	28

In case of **no crossing** of **connections**, you should **do nothing**.

You will continue receiving connections until you receive the command "**nexus**", at which point you must **print** what is **left** of the **arrays** and end the program.

Input

The input consists of several sequences:

- First you will receive **2 lines**, containing **sequences** of **integers**, separated by **spaces** – the arrays.
- On the next **several lines** you will receive the **pairs** of **connections** in the following format:
  `{firstArrayIndex}:{secondArrayIndex}|{firstArrayIndex}:{secondArrayIndex}`

- When you receive the command "**nexus**" the input must end.

Output

The output should consist of **2 lines**, containing what is left of the **arrays**, with its elements – **separated** by each other with a **comma** and a **space**.

Constraints

- The input will always be in valid format.
- The arrays will contain integers in **range [0, 10000]**.
- The arrays will **not necessarily** have the **same length**.
- The **indices** in the **connections** will always be **valid** and **inside** the arrays.
- Allowed time / memory: 100ms / 16MB.

Examples

Input	**Output**	**Comment**

`1 2 3 4 5 6 7 8 9 10`
`1 2 3 4 5 6 7 8 9 10`
`2:5|5:2`
`nexus`
	`19, 20, 25, 26, 27, 28`
	`19, 20, 25, 26, 27, 28`
		See the examples above.

`5 10 15 20 25 30`
`40 35 30 25 20 15 10 5`
`1:6|2:1`
`nexus`
	`75, 90, 95, 100`
	`110, 75`
		The **range** of the **elements** in the **second array** is **larger** than that of the **first array**.
		Naturally, **more elements** are **removed** from the second array.

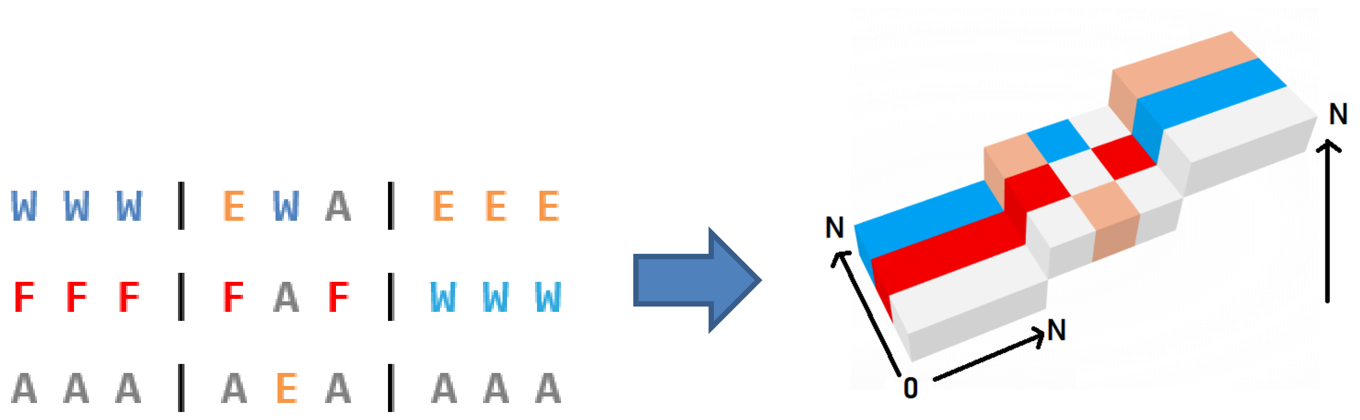
`9 5 10 4 5 6 7 10`
`3 3 3 4 5 6 7 8`
`0:1|1:0`
`0:1|1:0`
`0:1|1:0`
`nexus`
	`634, 637`
	`634, 635`	

4. Elemelons

If there is a watermelon, then there should be earthmelon, firemelon and airmelon. Introducing, the Elemelons! Doctor Sanity has a very unusual 3-D garden with elemenlons. The elemelons like to morph into each other when they are not observed, and Doc doesn't like that. He has called for your help in implementing an algorithm which helps him observe his surroundings while reaping elemelons so that they don't change.

You will be given **N** – the **dimensions** of the **3-D matrix** – the garden of Doctor Sanity, and the matrix itself afterwards, in a series of **N rows** containing the **columns** of **N matrices** – the **layers** of the 3-D matrix. The matrix will consist of the following symbols: **W** (**Watermelon**), **E** (**Earthmelon**), **F** (**Firemelon**), **A** (**Airmelon**).

For an example, let's color W (blue), E (orange), F (red) A (gray-white)
Check the examples below, for more info:



```
W W W | E W A | E E E
F F F | F A F | W W W
A A A | A E A | A A A
```

After you've successfully initialized the 3-D matrix, you'll start receiving **coordinates** – the cell that Doctor Sanity will **harvest** a melon from. **Upon harvesting** a melon, that cell in the 3-D matrix should be **set** to '**0**'. **All other melons**, **except** the **ones** in **direct sight** of Doc (**up**, **down**, **left**, **right**, **front**, **back** from the **currently harvested cell**), should **morph**.

The **morphing order** is: **Watermelon** (W) -> **Earthmelon** (E) -> **Firemelon** (F) -> **Airmelon** (A) -> **Watermelon** (W)… and so on.

Upon receiving the command "**Melolemonmelon**", the input sequence should end, and you should print the current state of the 3-D matrix in the same format, that you've received it from the input. Then you should end the program.

Input

The input consists of several lines:

- On the first line you will receive an integer number **N** – the size of the 3-D matrix.
- At the next **N** lines the **layers** of the **3-D matrix** are given (from **bottom** to **top**) as a sequence of **N** matrices separated by " **|** ".
- Afterwards, you will start receiving **coordinates** of a **cell** in the following format: "**{layer} {row} {column}**"
- When you receive the command "**Melolemonmelon**" the input sequence should end.

Output

As output you must print the current state of the 3-D matrix, in the same format as it came from the input – rows, containing the columns of N matrices – the layers of the cube (from **bottom** to **top**).

Constraints

- The integer **N** will be in **range [0, 50]**.
- The 3-D matrix will always be in a **valid format** and will **only contain** the following characters: '**W**', '**E**', '**F**', '**A**'.
- The **coordinates** for the **cell** to be **harvested**, will **always be inside** the 3-D matrix.
- Allowed time / memory: 100ms / 16MB.

Examples

Input	Output	Comment

`3`
`W W W | E W A | E E E`
`F F F | F A F | W W W`
`A A A | A E A | A A A`
`1 1 1`
`Melolemonmelon`
	`E E E | F W W | F F F`
	`A F A | F 0 F | E W E`
	`W W W | W E W | W W W`
		The cell at **layer 1**, **row 1**, **column 1** was **changed** to '**0**', as it was harvested.
		The yellow marked cells were in **direct sight** of Doctor Sanity, which is why the melons inside them **didn't morph**.
		All other cells' melons **morphed** into the **next** melon **in order**.

Input	Output

`4`
`A W F A | W W W W | W W W W | A A A A`
`W F W F | F F F F | A A A A | F F F F`
`A W E W | E E E E | A A A A | W E E W`
`A F W E | W W W W | W W W W | W W W W`
`1 1 1`
`2 3 2`
`3 1 3`
`Melolemonmelon`
	`F A E F | A F A A | A A A A | F F F E`
	`A W A E | W 0 W E | F E F E | E E W 0`
	`F A W A | W A W W | F F E F | A W W F`
	`F E A W | A A F A | A F 0 F | A A F A`

5. Grid Voyage

Doctor Sanity's pet pokemon – Slifer, is training his sense of direction with a game that Doc designed for him, called Grid Voyage. You should help Slifer, as he is not the smartest pokemon in existence. Try to write an algorithm which simulates the Grid Voyage game.

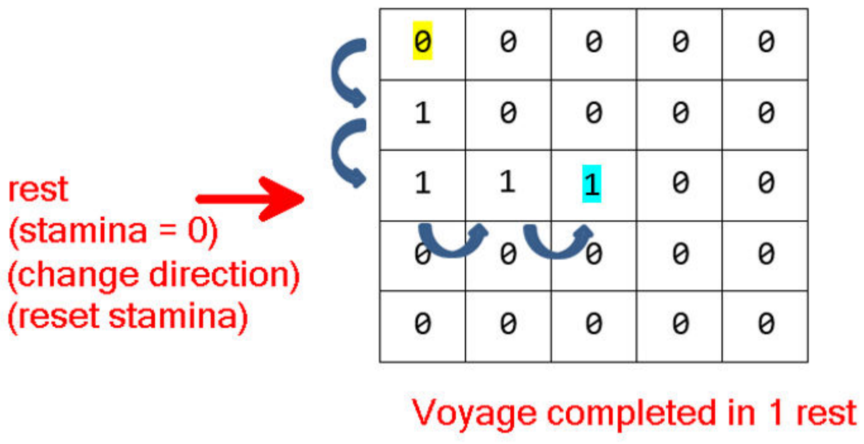
You will be given **N** – an **integer**. You must create a 2-D matrix of integers with **N rows** and **N columns**, each cell with value – **`0`**. You will then receive the **coordinates** of the **start point**.

Afterwards, you will start receiving lines, containing **coordinates** of a **destination point**, the **initial direction** (**`left`**, **`right`**, **up**, **down**) and **current stamina**. You must **reach** the **destination point** doing a grid-based movement, only changing directions when you run out of stamina.

- You consume **1 stamina** per **step** (per **cell movement**).
- When you run out of **stamina**, you **MUST change** the **direction** and **reset** your **stamina**.
- You must **increase** the **value** of **each cell** you **step on** with **1**.

Example: Initial position – **`[0, 0]`**.

First destination **`[2, 2]`**. Initial direction – **down**. Stamina – **2**.



Slifer is always looking for the **fastest path**, which means that he should **always move** in directions, **towards** his **next destination**.

Slifer is fat – he can **ONLY turn left** or **right**. He **CANNOT** completely **turn around** in a reversed direction.

Slifer possesses a bit of intellect – if he is in a **dilemma** (he can go several directions, and all of them are leading to his target), he will try to **turn LEFT first**.

- If he **cannot turn left** (not enough space in matrix, to make the needed steps before the next change in direction), he will try to **turn right**.
- If he **can** turn **neither left**, **nor right**, he will deem the current Voyage – **impossible** and return to his previous position. In this case, there should be **NO cells affected** by **increased value**.

If it **is possible** to reach the destination, you must print **how many rests** (how many times you've **ran out** of **stamina** and you've **changed direction**) Slifer took to reach it. If it is **NOT possible** to reach the destination, you must print "**`Voyage impossible`**".

When you receive the command "**`eastern odyssey`**", the input sequence should end, and you should print the **whole matrix**.

Input

The input consists of several input lines:

- On the **first** input line you will receive **N** – the **dimensions** of the 2-D matrix.
- On the **second** input line you will receive **X** and **Y**, separated by a **space** – the **initial position** of Slifer.
- On the next **several** input lines you will receive **coordinates** of a **destination point**, a **direction** and **stamina** in the following format: **`{destinationX} {destinationY} {direction} {stamina}`**
- When you receive the command "**`eastern odyssey`**", the input sequence should end.

Output

As output you must print:

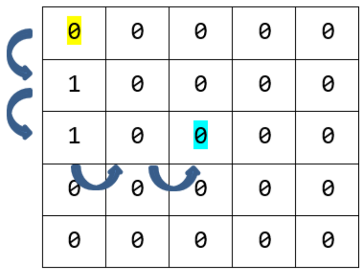
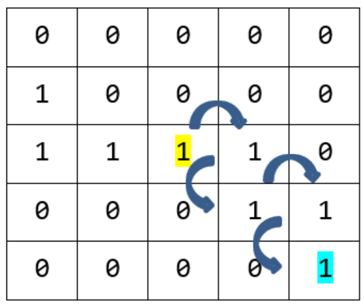
- For **every voyage**:
  - If it is **possible**, the number of **rests** it took to **reach** the **destination point**.
  - If it is **NOT possible** – you should print "**`Voyage impossible`**".
- At the end of the program:
  - The **whole matrix** – each **row** on a **new line**, each **column** separated by a **space**, from the others.

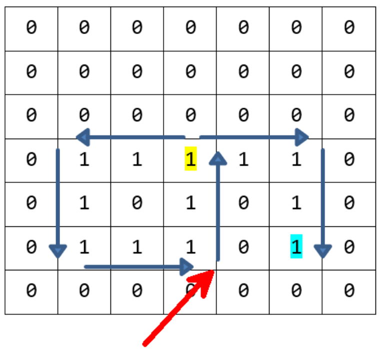
Constraints

- The integer **N** will be in **range [0, 50]**.
- The **coordinates** of the **initial point** and **destination points** will **always** be **VALID** and inside the matrix.
- The **directions** will **always** be **valid**.
- The **stamina** will always be in **range [1, 50]**.

Examples

Input	**Output**	**Comment**

`5`
`0 0`
`2 2 down 2`
`4 4 right 1`
`eastern odyssey`
	`1`
	`3`
	`0 0 0 0 0`
	`1 0 0 0 0`
	`1 1 1 1 0`
	`0 0 0 1 1`
	`0 0 0 0 1`
		First Voyage:
		
		Second Voyage:
		
		The **stamina** is **1** this time. We must **change direction** after **each step**.

`7`
`3 3`
`5 5 left 2`
`6 6 right 2`
`eastern odyssey`
	`5`
	`Voyage impossible`
	`0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0`
	`0 1 1 1 1 1 0`
	`0 1 0 1 0 1 0`
	`0 1 1 1 0 1 0`
	`0 0 0 0 0 0 0`
		
		Slifer reached a point where he can go both left and right. He always chooses left first.

6. Tri-Force

The TriForce is specific figure, formed by generating all possible triangles in a specific circle. You have been tasked to generate a TriForce by given parameters.

You will be given a **P** – a **perimeter** and a **R** – a **radius** of **circle**. Generate the sides of **all possible triangles** inscribed in a circle with the given **R** which have a **perimeter equal** to the **given one**.

NOTE: Consider only **integer** sides.

NOTE: A triangle with sides – a = 10, b = 12, c = 5, should be considered different from a triangle with sides a = 5, b = 12, c = 10.

NOTE: Generating should always be done from the side with the greatest possible value. See the examples for more info.

Input

The input will consist of 2 lines:

- On the **first** input line you will receive **P** – the **perimeter**.
- On the **second** input line you will receive **R** – the **radius** of the **circle**.

Output

The output will consist of several lines:

- As output you must print all possible triangles, following the rules above, in the following format:

  `{a}.{b}.{c}`

Constraints

- The perimeter **P** will be an integer (naturally, if all sides are integers) in **range [0, 30000]**.
- The radius **R** will be a floating-point number in **range [0, 15000]**.
- Allowed time / memory: 100ms / 16MB.

Examples

Input	Output
12 2.5	5.4.3 5.3.4 4.5.3 4.3.5 3.5.4 3.4.5
30 6.5	13.12.5 13.5.12 12.13.5 12.5.13

	5.13.12 5.12.13

7. Undefined

Have you ever heard of blockchain? Well, even if you didn't it is not a problem. In blockchain, its all about mining blocks. Mining a block is done by 2 nodes – each node is a type of business, but the 2 nodes (businesses) must have the same owner, and they must be connected only to each other.

You will receive N – an **integer**, which is the **amount** of **business owners**.

On the next **N lines** you will receive the owner's **initial** – a **letter** from the **alphabet**, and his **businesses** – which, will be **integers** – each **integer**, representing the corresponding business's **net worth**.

If **2 businesses** (a **pair** of **businesses**) are connected **ONLY** to **each other** and they have the **SAME owner**, they **WILL mine** a **block**. That block will have a **value** – equal to the **absolute value** of the **difference** between the 2 businesses' net worth.

You must generate a network of business owners and **pairs** of **businesses** in which you **mine** the **blocks** with the **highest summed up value**. However, note that, **NO business** should remain **disconnected**.

Input

The input will consist of several lines:

- On the **first** input line you will receive **N** – the **amount** of **business owners**.
- On the next **N lines** you will receive each owner's initial and businesses in the following format:

  `{owner} -> {business1}, {business2}, {business3}...`

Output

As output:

- You must print the owners, with each of their business pairs, in the following format:

`{owner} | {businessPair1First} <-> {businessPair1Second}, {businessPair2First}...`

  - Each owner must be printed on a **new line**.
  - The **owners** should be in **order** of **addition**.
  - The **businesses** should be **ordered** by **mined block value** in **descending order**.
  - If an owner **does not have** any pairs, you should just print "**none**".
- You must print the **leftover connections** (the businesses, that **did not mine** any **blocks**), if there are **any**, in the following format:

  `{owner}{business} <-> {otherOwner}{otherBusiness}`

  - The **leftover connections** must be **ordered** by the **sum** of each **2 businesses' net worth**, in **descending order**.
- You must print the **total mined block value**.

Constraints

- The integer **N – count** of **owners** will be in **range [0, 25]**.
- The **businesses' net worth** will be integers in **range [0, 100000]**.
- Each **owner** may be **given up** to **1000 businesses**.
- Allowed time / memory: 100ms / 16MB.

Examples

Input	Output	Comment

`3`
`A -> 60, 120, 40, 30`
`B -> 300, 4`
`C -> 50, 200, 220, 20`
	`A | 120 <-> 30, 60 <-> 40`
	`B | 300 <-> 4`
	`C | 220 <-> 20, 200 <-> 50`
	`756`	

`3`
`A -> 60, 120, 40, 30`
`B -> 300, 4, 4`
`C -> 50, 200, 220, 20, 5`
	`A | 120 <-> 30, 60 <-> 40`
	`B | 300 <-> 4`
	`C | 220 <-> 5, 200 <-> 20`
	`B4 <-> C50`
	`801`
		Notice how we have 2 more elements, one at B and one at 5 that are left-overs, after the pairs have been generated.
		We just pair them together and print the other pairs in the network, so that we mine the maximum block value.

`3`
`A -> 60, 120, 40, 30`
`B -> 300, 4, 4`
`C -> 50, 200, 220, 20`
	`A | 120 <-> 30`
	`B | 300 <-> 4`
	`C | 220 <-> 20, 200 <-> 50`
	`B4 <-> A60`
	`B4 <-> A40`
	`736`
		When you don't have another leftover element with which to pair one, you will need to **ruin** a **business pair**, and you must ruin the one that will bring you the least money, so that the network remains with the highest mined bock value.

8. Rooks

On a rectangular chess board with **X rows** and **Y columns**, **N rooks** should be placed in such a way, so that **each** of **them** is **attacked** by **at most 1** other **rook**. One rook is attacked by another rook, if they are placed on the **same row**, or on the **same column**, and there are **no other rooks between** them.

Write a program, which finds the **count** of **all possible ways** that these N rooks **can be placed** on the X / Y chess board, so that they **cover** the **conditions specified above**. Due to the fact, that the answer may be a **very big number**, always **print** the **remainder** of the **division** of the **actual count** with **`1,000,001`**.

Input

The input consists of 3 input lines:

- On the first line you will receive **X** – the **rows** of the chessboard
- On the second line you will receive **Y** – the **columns** of the chessboard
- On the third line you will receive **N** – the **count** of **rooks** that should be placed on the chessboard

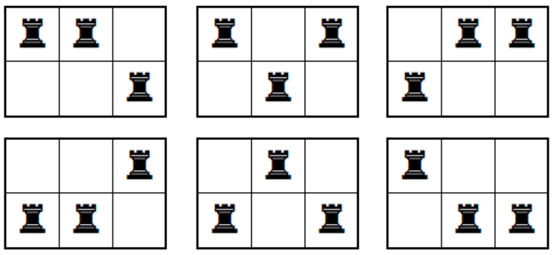
Output

The output should consist of a **single line**, containing the **remainder** of the **division** of the **desired count**, with **`1,000,001`**.

Constraints

- **X**, **Y** and **N** will be integers in **range [1, 100]**.
- Allowed working time / memory: to be defined.

Examples

Input	Output	Comment
4 6 2	276	There are only 2 rooks here and all ways they can be placed are valid. The answer is: (4 * 6) * (4 * 6 - 1) / 2 = 276.
2 3 3	6	
1 100 3	0	We cannot place 3 rooks on one row.
9 6 10	340200	
98 99 100	951454	The actual count is: 477162926599652378693202655573012530649120970956713215 413593915639833497991226651408072530190805115228714441 793077869034810249957308229677723184016455888816349376 99689344614655096884769587200000000000000000000000

9. Moneypoly

Mr. Moneybags's is a huge investor and a great businessman so it may come as no surprise that his favourite game is Moneypoly, it's kind of like Monopoly, but a bit different. The game's rules are very simple - the **only player is Mr.**

Moneybags and he starts with an **account balance of 0**. The playing area consists of **N indexed tiles**, with **Mr. Moneybags** being able to move from one tile to another, only if they have a **connection** between them.

Each tile has an integer value associated with it - the **investment result**, representing the **change to Mr. Moneybags's account balance** as result of his investment in the tile (a positive number meaning the investment paid off and Mr. Moneybags gained money and a negative number meaning the investment failed and Mr. Moneybags lost money).

Each **connection** between tiles has an integer value associated with it – the **investment time**, representing the amount of time it would take Mr. Moneybags to move to and acquire the tile on the other end of the **connection**.

Mr. Moneybags starts the game by stepping on the starting tile (tile with **index 0)**, each time he steps on(invests in) a tile he will modify his **account balance** with the tile's **investment result** and write the **index** of the tile in his **investment history**. After every investment, Mr. Moneybag can choose to either **end the game** or move to an adjacent tile that has a **connection** to the tile he is currently stepping on.

Mr. Moneybags believes in an optimized supply chain, so when the game begins he will **choose the set of connections with the smallest combined investment time, that would also provide him a path to every reachable tile,** and will then **only move using connections in that set**. One more thing, since Mr. Moneybags is good friends with the banks, while moving between tiles, he has the option to **declare bankruptcy**, setting his **account balance to 0**, and **deleting his investment history**, though **he still lands on the tile he is moving to** and has to modify his new account balance and investment history accordingly. Regardless of whether Mr. Moneybags has declared bankruptcy or not he will **never visit the same tile more than once**.

The **goal** is to find the **highest account balance** Mr. Moneybags can acquire and the **indexes of the investments** he had to take to get it.

Input

- On the first line you receive the number **N** – the **number of tiles**, with the tiles being **indexed** from **[0…N)**.
- On the next **N** lines you receive in ascending order the **index** of each tile with its associated **investment result** in the format **"{index} {investment result}"**.
- On the next lines, until the command **"end"** is received you will receive information about a **connection** in the format **"{tile1} {tile2} {investment time}"**.

Output

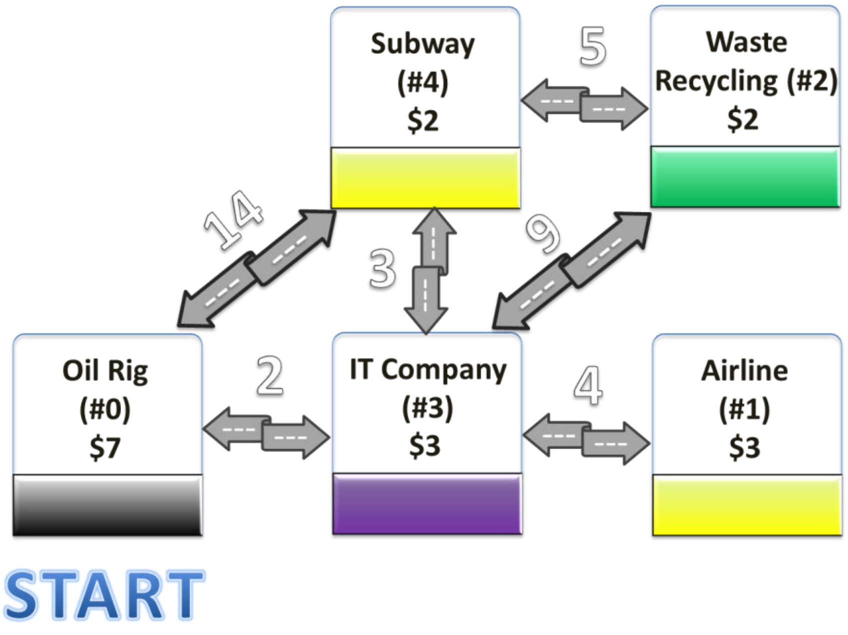
- On the first line of the output, you need to print the **highest account balance** Mr. Moneybags can get.
- On the second line print the **indexes of investments** he had to take as a **space separated sequence**, with the numbers sorted in **ascending order**.

Constraints

- The **Indexes** of the tiles will always be the numbers **[0…N-1]**.
- The **number of tiles N** will be between **[2…20 000]**.
- The **investment result** of each tile will be an integer in the range **[-1000…-1] ∪ [1…1000]**
- The **investment time** of each connection will be an integer in the range **[-1000…1000]**
- In case there are 2 or more connections, each of which can form its own set of connections with the lowest combined investment time, Mr. Moneybags will always take the connection ending in the tile with the lower numerical index (i.e. if we have a choice between 0->1 and 3->2, he will chose 0->1 – because 1 < 2). In case the 2 connections end in the same tile, he will always take the connection with the lower starting index (i.e. if we have a choice between 3->7 and 5->7, he will always choose 3->7, because 3 < 5).

- There will **never be more than one set of investments**, that Mr. Moneybags can take which grants the **highest account balance**.
- Mr. Moneybags will never visit a tile more than once.
- Mr. Moneybags will always start on the tile with **index 0**.
- Each connection can be traversed both ways.
- There will never be a connection between a tile and itself.
- Allowed time: **600 ms** Allowed memory: **32 MB**.

Examples

Input	Visualization	Output
5 0 7 1 3 2 2 3 3 4 2 2 3 9 0 3 2 3 4 3 1 3 4 0 4 14 4 2 5 end	 Subway (#4) $2 Waste Recycling (#2) $2 Oil Rig (#0) $7 IT Company (#3) $3 Airline (#1) $3 5 14 3 9 2 4 START	14 0 2 3 4

Comments

Mr. Moneybags start on the Oil Rig (#0) and add its investment result to his account balance:

Account Balance: 0 + 7 = 7

Investment History: #0

Now we have a choice between **ending the game**, going to **#4** or going to **#3**, we see we can make more profit if we go to a tile, so we decide to continue the game. Since Mr. Moneybags will only move through the set of connections with the smallest combined investment time, we need to first find that set. Looking at the playing area we can see that the set of connections with the smallest combined investment time, that also allows Mr. Moneybags to reach every tile on the field, is the following:

#0 - #3, #1 - #3, #2 - #4, #3 - #4

Having found the set of connections, we see that our only possible move is to **#3**, so we decide to go to **#3**.

Account Balance: 7 + 3 = 10

Investment History: #0, #3

Now again we have a choice, we can **end the game**, go to **#4**, go to **#2** or go to **#1**. Since again we can make more profit by going to a tile, we decide to continue the game. We see that **#1** would give us the most money, however since Mr. Moneybags never steps on the same tile twice, if we go to it we would be stuck, so we decide **NOT** to go to **#1**. The only remaining choice that is part of the set is **#4**, so we decide to go to **#4**.

Account Balance: 10 + 2 = 12

Investment History: #0, #3, #4

Again we have a choice, we can end the game or go to **#2**, since again we can increase our account balance by going to **#2** and the connection is in the set, we decide to continue the game and go to **#2**.

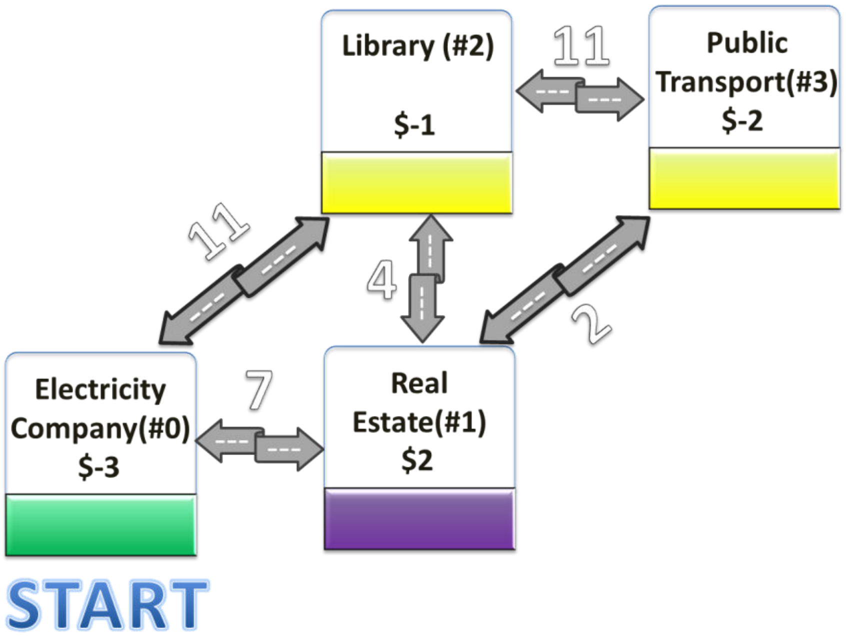
Account Balance: 12 + 2 = 14

Investment History: #0, #2, #3, #4

We have no more tiles we can visit, so we end the game, leading us to the answer of:

Account Balance: 14

Investment History: #0, #2, #3, #4

Input	Visualization	Output
4 0 -3 1 2 2 -1 3 -2 1 0 7 3 2 11 1 2 4 1 3 2 2 0 11 end	 Library (#2) $-1 11 Public Transport(#3) $-2 11 4 2 Electricity Company(#0) $-3 7 Real Estate(#1) $2 START	2 1

Comments

Mr. Moneybags starts on the Electricity Company (#0) , it turns out the investment was bad so we lose money:

Account Balance: 0 - 3 = -3

Investment History: #0

Now we have a choice between **ending the game**, going to **#1** or going to **#2**, since going to **#1** will make us money and the connection to it also is part of the set with the smallest combined investment time, we decide to continue playing and go to **#1**. Before stepping on **#1**, Mr. Moneybags uses his connections to the banks, to declare bankruptcy and have his account balance and investment history reset.

Account Balance: 0

Investment History:

After declaring bankruptcy, Mr. Moneybags lands on **#1** and we modify his account balance and investment history accordingly:

Account Balance: 0 + 2 = 2

Investment History: #1

Now again we have a choice, we can **end the game**, go to **#2** or go to **#3**. Since there doesn't seem to be any more ways to increase our account balance, we decide it's time to end the game. Mr. Moneybags' final account balance and investment history are:

Account Balance: 2

Investment History: #1

10. Plants

An interesting conflict between two species is taking place on the newly renovated Gaf Inatiev Boulevard (*any similarities to real places and events are purely coincidental… honest!*). What makes it even more interesting is the two species are… plants! Exciting, right? Well, one of them is actually a fungus, so not really a plant, but the other one is a tree, so – close enough.

The battle is happening on the new grates around the trees on the boulevard – around each tree there are channels for its roots to grow. Each tree's grate has between **1** and **359** channels, starting from the tree and going outwards in a radial pattern. The lengths of all the channels are the same (i.e. if you have e.g. 359 channels with a length of 5cm, this essentially creates a circle with a radius of 5cm).

The tree's **roots** start growing **from the center** of the grate **towards the outside**, but a species of **fungus** has started growing on the **outside**, **towards the center**. The tree and the fungus are **incompatible** – the roots of the tree can't grow over the fungus, and the fungus cannot grow over the tree. They can reach each other, but **not overlap**.

The **tree** grows only during the **day**, the **fungus** grows only during the **night**. During a single **day**, the tree can only **grow along a single channel**. The fungus can also grow only along a **single channel during the night** (it does not have to be in the same channel in which the tree grew during the day). Both the fungus and the tree grow by integer increments.

Both the tree and the fungus **must grow during each day/night**, and they can only grow in **parts** of the channels which are **not occupied**. If the tree **can't grow** during the day, it **dies**. If the fungus **can't grow** during the night, it **dies**. Hence the conflict – the **first one that fails to grow** (due to all the space in the channels being occupied) **dies**. *This grate ain't big enough for both of 'em, wild-west style*.

The tree and the fungus have evolved to **compete optimally** in this conflict. Each of them will always **grow optimally**, **if possible**. This means that they will choose such a length of **growth along a single channel**, that their "opponent" is **either prevented from growing**, or forced into a **position where there is no possible optimal move** (i.e. a position that eventually will lead to them not being able to grow, if the current player continues to grow optimally).

You are examining a specific grate during the beginning of **the day**. You see the **number** of channels, the **length** of each tree **root along each channel**, the **length** of the **fungus along each channel**, and the **radius** of the channels (i.e. the radius of the grate).

Your task is to write a program which, given the above information, determines what the optimal growth for the tree is during this day. The answer should contain the **number of a channel**, **as ordered in the input**, **starting from**

`0`, and the **length** of the growth. If there are **multiple** optimal options for the growth, choose the one in the channel with the **lowest number**.

If there is **no optimal growth**, the program should **indicate that** (see the output description below).

Input

The first line of the standard input will contain the integer number **N** – the number of channels.

Each of the next **N** lines will contain two integer numbers, separated by a single space – the **length of the root** and the **length of the fungus** in that channel (starting from channel **`0`** and ending in channel **`N-1`**).

The last line of the standard input will contain the integer number **R** – the radius of the grate, i.e. the length of each channel.

Output

The output should consist of a single line on the standard output.

If there is a optimal growth option, the output should be in the format:

`grow I by L`

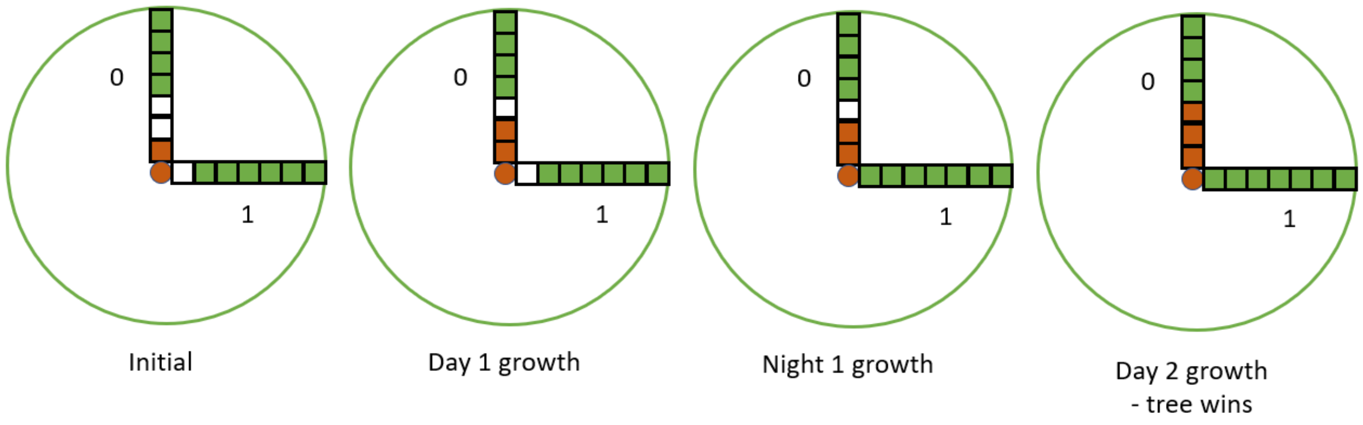
where **`I`** is the number of the channel (as ordered in the input) and **`L`** is the length of the growth. If there are multiple options, print the one with the lowest channel number**.** For example, if the optimal growth options are channel **3** by a length of **5** and channel **1** by a length of **7**, the output should be **`"grow 1 by 7"`**.

If there is no optimal growth option, print three dashes:

`---`

Illustration

Here is a situation with a grate, which has only 2 channels, with a radius of 7 (the grate channels are divided into sections to illustrate their length of 7, the brown sections are tree roots, the green sections are fungus):



In the initial state, channel 0 has a root with a length of 1 and a fungus length of 4. Channel 1 has a root length of 0 and a fungus length of 6. Since this is the beginning of the day, the tree will grow. The optimal growth for day 1 is in channel 0 by 1. That way the fungus is forced to grow by 1 either in channel 0 or channel 1 during night 1 (note that this is a bad position). Wherever the fungus grows during night 1, it will fill up the channel leaving only 1 channel free – the tree grows in that in day 2, so during night 2 the fungus can't grow and will die.

Restrictions

`0 < N < 360`

`0 < R < 1001`

No root or fungus length inside a channel will be longer than **R**.

There is **no limit to the length** the tree/fungus can grow during a day/night, but they may only grow inside a **single channel** per day/night.

Each root is a single segment and each fungus growth inside a channel is a single segment – *you can't "skip over" the sections in the illustration above*.

N and **R** will be integers.

Examples

Input	Output	Explanation

`2`
`1 4`
`0 6`
`7`
	`grow 0 by 1`	See the illustration

`3`
`5 1`
`0 7`
`2 3`
`10`
	`grow 1 by 2`	The tree grows in channel 1, by a length of 2. This is the only optimal growth in this situation, and it leaves the fungus in a situation where regardless of its growth, the tree can force it into a similar situation until the last growth remains for the tree

`5`
`1 1`
`0 2`
`2 0`
`0 2`
`2 0`
`4`
	`grow 0 by 2`	There are multiple optimal growths for the tree here, but we pick the one with the lowest channel number – 0

`4`
`0 2`
`2 0`
`0 2`
`2 0`
`4`
	`---`	The tree can't survive if the fungus grows optimally. There is an even number of equally empty channels. If the tree grows fully into a channel, the fungus will grow fully in another and due to the even number of repetitions will be the last to grow. If the tree grows by 1 in any channel, the fungus will grow by 1 in any other channel and the tree will be placed in effectively the same position.