Exercise: Lists Advanced

Problems for exercise and homework for the Python Fundamentals Course @SoftUni.
Submit your solutions in the SoftUni judge system at https://judge.softuni.org/Contests/1731.

1. Which Are In?

You will be given **two sequences** of strings, separated by "**, **". Print a **new list** containing only the **strings** from the **first input line,** which are **substrings** of **any string** in the **second input line**.

Example

Input	Output

`arp, live, strong`
`lively, alive, harp, sharp, armstrong`
	`['arp', 'live', 'strong']`

`tarp, mice, bull`
`lively, alive, harp, sharp, armstrong`
	`[]`

2. Next Version

You are fed up with changing the version of your software manually. Instead, you will create a little script that will make it for you.

You will be given a string representing the **version** of your software in the format: "**`{n1}.{n2}.{n3}`**". Your task is to **print** the **next version.** For example, if the current version is "**`1.3.4`**", the next version will be "**`1.3.5`**".

The only **rule** is that the numbers cannot be **greater than 9**. If it happens, set the **current number to 0** and **increase the previous number**. For more clarification, see the examples below.

Note: there will be no case in which the first number will become greater than 9.

Example

Input	Output
`1.2.3`	`1.2.4`
`1.3.9`	`1.4.0`
`3.9.9`	`4.0.0`

3. Word Filter

Using **comprehension**, write a program that receives some **text**, separated by **space**, and take only those words whose length is **even**. Print each word on a new line.

Examples

Input	Output

`kiwi orange banana apple`	`kiwi`
	`orange`
	`banana`

`pizza cake pasta chips`	`cake`

4. Number Classification

Using a **list comprehension**, write a program that receives **numbers**, separated by comma and space "**, **", and prints all the **positive**, **negative**, **even,** and **odd** numbers on separate lines as shown below.

Note: Zero is counted for a positive number

Examples

Input	Output

`1, -2, 0, 5, 3, 4, -100, -20, 12, 19, -33`	`Positive: 1, 0, 5, 3, 4, 12, 19`
	`Negative: -2, -100, -20, -33`
	`Even: -2, 0, 4, -100, -20, 12`
	`Odd: 1, 5, 3, 19, -33`

`1, 2, 53, 2, 21`	`Positive: 1, 2, 53, 2, 21`
	`Negative:`
	`Even: 2, 2`
	`Odd: 1, 53, 21`

5. Office Chairs

You are a facility manager at a large business center. One of your responsibilities is to check if each conference room in the center has enough chairs for the visitors.

On the first line, you will be given an integer **n** representing **the number of rooms in the business center**. On the following **n lines** for each room, you will receive information about the **chairs** in the room and the **number of visitors**. Each **chair** will be presented with the char "**X**". Next, there will be a **single space** and the number of visitors at the end. For example: "**XXXXX 4**" (**5 chairs** and **4 visitors**).

Keep track of the free chairs:

- If there are **not enough chairs** in a specific room, print the following message: "**`{needed_chairs_in_room} more chairs needed in room {number_of_room}`**". The rooms start from 1.
- Otherwise, print: "**`Game On, {total_free_chairs} free chairs left`**".

Example

Input	Output

`4`
`XXXX 4`
`XX 1`
`XXXXXX 3`
`XXX 3`
	`Game On, 4 free chairs left`

`3`
`XXXXXXX 5`
`XXXX 5`
`XXXXXX 8`
	`1 more chairs needed in room 2`
	`2 more chairs needed in room 3`

6. Electron Distribution

You are a mad scientist, and you have decided to play with electron distribution among atom shells. The basic idea of electron distribution is that electrons should fill a shell until it holds the maximum number of electrons.

You will receive a single integer – the **number of electrons**. Your task is to **fill shells until there are no more electrons left**. The **rules** for electron distribution are as follows:

- The maximum number of electrons in a shell can be **$2n^2$**, where **n** is the **position** of a **shell** (**starting from 1**). For example, the maximum number of electrons in the 3rd shield can be **$2*3^2$ = 18**.
- You should start **filling** the shells from the **first one** at the first position.
- If the electrons are enough to **fill** the **first** shell, the left **unoccupied electrons** should fill the **following** shell and so on.

In the end, **print a list with the filled shells**.

Example

Input	Output
10	[2, 8]
44	[2, 8, 18, 16]

7. Group of 10's

Write a program that receives a **sequence of numbers** (a string containing **integers** separated by "**,** ") and **prints** the **numbers sorted into lists** of **10's** in the format "**Group of {group}'s: {list_of_numbers}**".

Examples:

- The numbers **2, 8, 4, and 10** fall into the group of **10's**.
- The numbers **13, 19, 14, and 15** fall into the group of **20's**.

For more clarification, see the examples below.

Example

Input	Output
8, 12, 38, 3, 17, 19, 25, 35, 50	Group of 10's: [8, 3] Group of 20's: [12, 17, 19] Group of 30's: [25] Group of 40's: [38, 35] Group of 50's: [50]
1, 3, 3, 4, 34, 35, 25, 21, 33	Group of 10's: [1, 3, 3, 4] Group of 20's: [] Group of 30's: [25, 21] Group of 40's: [34, 35, 33]

Hints

- **Keep track of the group** using a variable to store its **max value.**
- Create a **loop** and **filter the elements** that are less than or equal to the group boundary and **remove** them from the **original list.**
- **Increase** the **boundary by 10.**
- **Loop until** the given **list is empty.**

8. Decipher This!

You are given a **secret message** you should **decipher**. To do that**,** you need to know that **in each word**:

- the **second** and the **last letter** are **switched** (e.g., Holle means Hello)
- the **first letter** is **replaced** by its **character code** (e.g., 72 means H)

Example

Input	Output
72olle 103doo 100ya	Hello good day
82yade 115te 103o	Ready set go

9. * Moving Target

You are at the shooting gallery again, and you need a program that helps you keep track of moving targets. On the first line, you will receive a **sequence of targets with their integer values**, split by a **single space**. Then, you will start receiving **commands for manipulating the targets** until the **"End"** command. The commands are the following:

- **"Shoot {index} {power}"**
  - Shoot the target at the index **if it exists** by **reducing** its **value** by the **given power** (**integer value**). A target is considered **shot** when **its value reaches 0**.
  - Remove the target **if it is shot**.
- **"Add {index} {value}"**
  - Insert a target with the received value at the received **index if it exists**. If not, print: **"Invalid placement!"**
- **"Strike {index} {radius}"**
  - Remove the **target at the given index** (**if such exist**) and the **ones before and after it depending on the radius.**
  - If any of the **indices** in the range is **invalid**, print: **"Strike missed!"** and skip this command.

Example: "Strike 2 2"

	{radius}	{radius}	{strikeIndex}	{radius}	{radius}		

- **"End"**
  - Print the sequence with targets in the following format:
    "{target$_1$}|{target$_2$} … |{target$_n$}"

Input / Constraints

- On the **first line,** you will receive **the sequence of targets** – **integer values [1-10000]**.
- On the **following lines,** until the **"End"**, you will be receiving the command described above – **strings**.
- There will never be a case when the **"Strike"** command would empty the whole sequence.

Output

- Print the appropriate message in case of the "**Strike**" command if necessary.
- In the end, print the sequence of targets in the format described above.

Examples

Input	Output	Comments

`52 74 23 44 96 110`
`Shoot 5 10`
`Shoot 1 80`
`Strike 2 1`
`Add 22 3`
`End`
	`Invalid placement!`
	`52|100`
		The first command is "**Shoot**", so we reduce the target on **index 5**, which is valid, with the given **power** – **10**.
		Then we receive the same command, but we need to reduce the target on the 1st index, with power 80. The value of this target is 74, so it is considered shot, and we **remove** it.
		Then we receive the "**Strike**" command on the 2nd index, and we need to check if the range with radius 1 is valid:
		52 23 44 96 100
		And it is, so we **remove** the targets.
		At last, we receive the "**Add**" command, but the index is **invalid,** so we print the appropriate **message**, and in the end, we have the following result:
		52|100

`47 55 85 78 99 20`
`Shoot 1 55`
`Shoot 8 15`
`Strike 2 3`
`Add 0 22`
`Add 2 40`
`Add 2 50`
`End`
	`Strike missed!`
	`22|47|50|40|85|78|99|20`	

10. * Heart Delivery

Valentine's Day is coming, and Cupid has minimal time to spread some love across the neighborhood. Help him with his mission!

You will receive a **string** with **even integers,** separated by a **"@"** - this is our neighborhood. After that, a series of **Jump** commands will follow until you receive **"Love!"**. Every house in the neighborhood needs a certain number of **hearts** delivered by Cupid so it can celebrate Valentine's Day. The integers in the neighborhood indicate those needed hearts.

Cupid starts at the position of the **first house** (index 0) and must jump by a **given length.** The jump commands will be in this format: **"Jump {length}"**.

Every time he jumps from one house to another, the needed hearts for the visited house are **decreased by 2**:

- If the needed hearts for a certain house become **equal to 0**, print on the console **"Place {house_index} has Valentine's day."**
- If **Cupid** jumps to a house where the needed hearts are **already 0,** print on the console **"Place {house_index} already had Valentine's day."**
- Keep in mind that **Cupid** can have a **larger jump length** than the **size of the neighborhood,** and if he does jump **outside** of it, he should **start** from the **first house** again (index 0)

For example, we are given this neighborhood: 6@6@6. Cupid is at the start and jumps with a length of 2. He will end up at index 2 and decrease the needed hearts by 2: [6, 6, 4]. Next, he jumps again with a length of 2 and goes outside the neighborhood, so he goes back to the first house (index 0) and again decreases the needed hearts there: [4, 6, 4].

Input

- On the first line, you will receive a **string** with **even integers** separated by **`"@"`** – the neighborhood and the number of hearts for each house.
- On the next lines, until `"Love!"` is received, you will be getting jump commands in this format: **`"Jump {length}"`**.

Output

In the end, print **Cupid's last position** and whether his mission was successful or not:

- **`"Cupid's last position was {last_position_index}."`**
- If **each house** has had Valentine's day, print:
    - **`"Mission was successful."`**
- If **not,** print the **count** of all houses that **didn't** celebrate Valentine's Day:
    - **`"Cupid has failed {houseCount} places."`**

Constraints

- The **neighborhood's** size will be in the range [1…20]
- Each **house** will need an **even number** of hearts in the range [2 … 10]
- Each **jump length** will be an integer in the range [1 … 20]

Examples

Input	Output	Comments

`10@10@10@2`
`Jump 1`
`Jump 2`
`Love!`
	`Place 3 has Valentine's day.`
	`Cupid's last position was 3.`
	`Cupid has failed 3 places.`
		Jump 1 ->> [10, 8, 10, 2]
		Jump 2 ->> [10, 8, 10, 0] so we print "Place 3 has Valentine's day."
		The following command is "Love!" so we print Cupid's last position and the outcome of his mission.

`2@4@2`
`Jump 2`
`Jump 2`
`Jump 8`
`Jump 3`
`Jump 1`
`Love!`
	`Place 2 has Valentine's day.`
	`Place 0 has Valentine's day.`
	`Place 0 already had Valentine's day.`
	`Place 0 already had Valentine's day.`
	`Cupid's last position was 1.`
	`Cupid has failed 1 places.`	

11. * Inventory

As a young traveler, you gather items and craft new items.

You will receive a journal with some Collecting items, separated with `", "` (comma and space). After that, until receiving **`"Craft!"`** you will be receiving different commands.

Commands (split by **" - "**):

- **"Collect - {item}"** – Receiving this command, you should add the given item to your inventory. If the item already **exists**, you should **skip** this line.
- **"Drop - {item}"** – You should remove the item from your inventory **if it exists**.
- **"Combine Items - {oldItem}:{newItem}"** – You should check if the **old item exists**. If so, **add** the new item **after** the **old one**. Otherwise, **ignore** the command.
- **"Renew – {item}"** – If the given item exists, you should change its position and **put it last** in your inventory.

Output

After receiving **"Craft!"** print the items in your inventory, separated by **", "** (comma and space).

Examples

Input	Output
Iron, Wood, Sword Collect - Gold Drop - Wood Craft!	Iron, Sword, Gold
Iron, Sword Drop - Bronze Combine Items - Sword:Bow Renew - Iron Craft!	Sword, Bow, Iron