Exercise: Lists Advanced

This document defines the exercises for the "Python Fundamentals" course at @SoftUni Global

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 takes 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 as 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 3^{rd} 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. *Anonymous Threat

Anonymous has created a hyper cyber virus, which steals data from the CIA. The virus is known for its innovative and unbelievably clever merging and dividing data into partitions. As the lead security developer in the CIA, you have been tasked to analyze the software of the virus and observe its actions on the data.

You will receive a **single input line** containing **strings,** separated by **spaces**. The strings may contain **any ASCII** character except **whitespace**. Then you will begin receiving commands in one of the following formats:

- **`merge {startIndex} {endIndex}`**
- **`divide {index} {partitions}`**

Every time you receive the **`merge command`**, you must merge all elements from the **`startIndex`** to the **`endIndex`**. In other words, you should concatenate them.
Example: **`{abc, def, ghi} -> merge 0 1 -> {abcdef, ghi}`**

If **any** of the **given indexes** is **out of the array**, you must take **only** the **range** that is **inside** the **array** and **merge** it.

Every time you receive the **`divide command`**, you must **divide** the **element** at the **given index** into **several small substrings** with **equal length**. The **count** of the **substrings** should be **equal** to the **given partitions**.

Example: **`{abcdef, ghi, jkl} -> divide 0 3 -> {ab, cd, ef, ghi, jkl}`**

If the string **cannot** be **exactly divided** into the **given partitions**, **make all partitions except** the **last** with **equal lengths** and make the **last one** - **the longest**.

Example: **`{abcd, efgh, ijkl} -> divide 0 3 -> {a, b, cd, efgh, ijkl}`**

The **input ends** when you receive the command "**`3:1`**". At that point, you must print the **resulting elements**, **joined** by a **space**.

Input

- The **first input line** will contain the **array** of **data**.
- On the **next several input** lines, you will **receive commands** in the **format specified above**.
- The **input ends** when you receive the command "**`3:1`**".

Output

- As output, you must print a single line containing the elements of the array, **joined** by a **space**.

Constraints

- The **strings** in the **array** may contain any **ASCII character** except **whitespace**.

- The **`startIndex`** and the **`endIndex`** will be in the **range [-1000...1000]**.
- The **`endIndex`** will **always** be **greater** than the **`startIndex`**.
- The **`index`** in the **`divide` command** will **always** be **inside** the array.
- The **`partitions`** will be in the **range [0...100]**.
- Allowed working **time/memory**: **100ms / 16MB**.

Examples

Input	Output

`Ivo Johny Tony Bony Mony`
`merge 0 3`
`merge 3 4`
`merge 0 3`
`3:1`
	`IvoJohnyTonyBonyMony`

`abcd efgh ijkl mnop qrst uvwx yz`
`merge 4 10`
`divide 4 5`
`3:1`
	`abcd efgh ijkl mnop qr st uv wx yz`

10. *Pokemon Don't Go

Ely likes to play Pokemon Go a lot. But Pokemon Go bankrupted… So the developers made Pokemon Don't Go out of depression. And so Ely now plays Pokemon Don't Go. In Pokemon Don't Go, when you walk to a certain pokemon, those closest to you naturally get further, and those further from you, get closer.

You will receive a **sequence** of **integers**, separated by **spaces** - the distances to the pokemon. Then you will begin **receiving integers**, which will **correspond** to **indexes** in **that sequence**.

When you **receive** an **index**, you must **remove** the **element** at **that index** from the **sequence** (as if you've captured the pokemon).

- You must **increase** the **value** of **all elements** in the sequence that are **less** or **equal** to the **removed element** with the **value** of the **removed element**.
- You must **decrease** the **value** of **all elements** in the sequence that are **greater** than the **removed element** with the **value** of the **removed element**.

If the **given index** is **less** than **0**, **remove** the **first element** of the **sequence**, and **copy** the **last element** to its place.

If the **given index** is **greater** than the **last index** of the **sequence**, **remove** the **last element** from the sequence, and **copy** the **first element** to its place.

The **increasing** and **decreasing** elements should also be done in these cases. The **element** whose value you should use is the **removed** element.

The program **ends** when the **sequence** has **no elements** (there are no pokemon left for Ely to catch).

Input

- On the **first line** of input, you will receive a **sequence** of **integers**, **separated** by **spaces**.
- On the **next several** lines, you will receive **integers** - the **indexes**.

Output

- When the program ends, you must print the **summed value** of **all removed elements**.

Constraints

- The input data will consist **only** of **valid integers** in the **range [-2.147.483.648…2.147.483.647]**.

Examples

Input	Output	Comments
4 5 3 1 1 0	14	The **array** is **{4, 5, 3}**. The index is **1**. We **remove 5**, and we **increase all the lower** ones and **decrease all the higher** ones. In this case, there are **no higher** than **5**. The result is **{9, 8}**. The **index** is **1**. So we remove **8** and **decrease all the higher** ones. The result is **{1}**. The **index** is **0**. So we remove **1**. There are **no elements left**, so we print the **sum** of **all removed elements**. **5 + 8 + 1 = 14**.
5 10 6 3 5 2 4 1 1 3 0 0	51	**Step 1**: **{11, 4, 9, 11}** **Step 2**: **{22, 15, 20, 22}** **Step 3**: **{7, 5, 7}** **Step 4**: **{2, 2}** **Step 5**: **{4, 4}** **Step 6**: **{8}** **Step 7**: **{} (empty).** **Result** = **6 + 11 + 15 + 5 + 2 + 4 + 8 = 51**.

11. *SoftUni Course Planning

Help plan the next Programming Fundamentals course by keeping track of the lessons that will be included in the course and all the exercises for the lessons. Before the course starts, there are some changes to be made.

On the **first input line**, you will receive the initial schedule of lessons and exercises that will be part of the next course, separated by a comma and a space **", "**. Until you receive the **"course start"** command, you will be given some **commands to modify the course schedule**.

The **possible commands** are:

- **"Add:{lessonTitle}"** - **add the lesson to the end** of the schedule if it **does not exist.**
- **"Insert:{lessonTitle}:{index}"** - **insert** the lesson to the **given index**, if it **does not exist.**
- **"Remove:{lessonTitle}"** - remove the lesson, **if it exists.**
- **"Swap:{lessonTitle}:{lessonTitle}"** - **swap the position** of the two lessons **if they exist.**
- **"Exercise:{lessonTitle}"** - add Exercise in the schedule right after the lesson index**, if the lesson exists and there is no exercise already**, in the following format **"{lessonTitle}-Exercise"**. **If the lesson doesn't exist**, **add the lesson** at the end of the course schedule**, followed by the Exercise**.

Note: Each time you Swap or Remove a lesson, you should do the same with the Exercises, if there are any following the lessons.

Input / Constraints

- On the first line - the initial schedule lessons - strings, separated by comma and space **", "**.
- Until **"course start"** you will receive commands in the format described above.

Output

- Print the whole course schedule, each lesson on a new line with its number (index) in the schedule: **"{lesson index}.{lessonTitle}".**
- Allowed working **time** / **memory**: **100ms** / **16MB.**

Examples

Input	Output	Comment
Data Types, Objects, Lists Add:Databases Insert:Arrays:0 Remove:Lists course start	1.Arrays 2.Data Types 3.Objects 4.Databases	We receive the initial schedule. Next, we add the Databases lesson, because it doesn't exist. We Insert at the given index lesson Arrays because it's not present in the schedule. After receiving the last command and removing lesson Lists, we print the whole schedule.
Arrays, Lists, Methods Swap:Arrays:Methods Exercise:Databases Swap:Lists:Databases Insert:Arrays:0 course start	1.Methods 2.Databases 3.Databases-Exercise 4.Arrays 5.Lists	We swap the given lessons because both exist. After receiving the Exercise command, we see that such a lesson doesn't exist, so we add the lesson at the end, followed by the exercise. We swap Lists and Databases lessons, and the Databases-Exercise is also moved after the Databases lesson. We skip the next command because we already have such a lesson in our schedule.