Exercise: Lists

Problems for exercise and homework for the "C# Fundamentals" course @ SoftUni
You can check your solutions in Judge

1. Train

On the first line, we will receive a **list of wagons** (integers). Each **integer** represents the **number of passengers** that are currently in each wagon of the passenger train. On the next line, you will receive the **max capacity of a wagon**, represented as a **single integer**. **Until** you receive the **"end"** command, you will be receiving two types of input:

- **Add {passengers}** – add a wagon to the end of the train with the given number of passengers.
- **{passengers}** – **find a single wagon** to fit all the incoming passengers (starting from the first wagon).

In the end, **print** the final state of the train (all the wagons separated by a space).

Example

Input	Output
32 54 21 12 4 0 23 75 Add 10 Add 0 30 10 75 end	72 54 21 12 4 75 23 10 0
0 0 0 10 2 4 10 Add 10 10 10 10 8 6 end	10 10 10 10 10 10 10

2. Change List

Create a program, that **reads a list of integers** from the console and receives **commands** to **manipulate the list**.

Your program may receive the following **commands**:

- **Delete {element}** – delete all elements in the array, which are equal to the given element.
- **Insert {element} {position}** – insert the element at the given position.

You should **exit the program** when you receive the **"end"** command. Print all numbers in the array, **separated by a single whitespace**.

Examples

Input	Output
1 2 3 4 5 5 5 6 Delete 5	1 10 2 3 4 6

Insert 10 1 Delete 5 end	
20 12 4 319 21 31234 2 41 23 4 Insert 50 2 Insert 50 5 Delete 4 end	20 12 50 319 50 21 31234 2 41 23

3. House Party

Create a program that keeps track of the guests that are going to a house party. On the first line, of input you are going to receive **the number of commands that will follow**.

On the next lines, you are going to receive some of the following: **"{name} is going!"**

- You have to **add the person, if they are not on the guestlist already.**
- If **the person is on the list** print the following to the console: **"{name} is already in the list!"**

"{name} is not going!"

- You have to remove the person, if they are on the list.
- If not, print out: **"{name} is not in the list!"**

Finally, print all of the guests, each on a new line.

Examples

Input	Output
4 Allie is going! George is going! John is not going! George is not going!	John is not in the list! Allie
5 Tom is going! Annie is going! Tom is going! Garry is going! Jerry is going!	Tom is already in the list! Tom Annie Garry Jerry

4. List Operations

The first input line will hold a list of **integers**. Until we receive the **"End"** command, we will be given **operations** we have to apply to the list.

The **possible commands** are:

- **Add {number}** – add the given number to the end of the list
- **Insert {number} {index}** – insert the number at the given index
- **Remove {index}** – remove the number at the given index
- **Shift left {count}** – first number becomes last. This has to be repeated the specified number of times
- **Shift right {count}** – last number becomes first. To be repeated the specified number of times

Note: the index given may be outside of the bounds of the array. In that case print: **`"Invalid index".`**

Examples

Input	Output
1 23 29 18 43 21 20 Add 5 Remove 5 Shift left 3 Shift left 1 End	43 20 5 1 23 29 18
5 12 42 95 32 1 Insert 3 0 Remove 10 Insert 8 6 Shift right 1 Shift left 2 End	Invalid index 5 12 42 95 32 8 1 3

5. Bomb Numbers

Create a program that reads a sequence of numbers and a special **bomb** number **holding a certain power**. Your task is to **detonate every occurrence of the special bomb number** and according to its power **the numbers to its left and right**. The bomb power refers to how many numbers to the left and right will be removed, no matter their values. Detonations are performed **from left to right** and all the detonated numbers **disappear**. Finally, **print the sum of the remaining elements** in the sequence.

Examples

Input	Output	Comments
1 2 2 4 2 2 2 9 4 2	12	The special number is **4** with power **2**. After detonation, we are left with the sequence `[1, 2, 9]` with sum 12.
1 4 4 2 8 9 1 9 3	5	The special number is **9** with power **3.** After detonation, we are left with the sequence `[1, 4]` with sum 5. Since the 9 has only 1 neighbor from the right we remove just it (one number instead of 3).
1 7 7 1 2 3 7 1	6	Detonations are performed from left to right. We cannot detonate the second occurrence of **7,** because it's already destroyed by the first occurrence. The numbers `[1, 2, 3]` survive. Their sum is 6.
1 1 2 1 1 1 2 1 1 1 2 1	4	The red and yellow numbers disappear in two sequential detonations. The result is the sequence `[1, 1, 1, 1]`. Sum = 4.

6. Cards Game

You will be given two hands of cards, which will be represented by **integers**. Assume each one is held by a different player. You have to **find** which one has the **winning deck**. You start from the beginning of both hands of cards. **Compare** the cards from the first deck to the cards from the second deck. The player, **who holds the more powerful card** on the current iteration, takes both cards and puts them at the back of his hand - the second player's card is placed last and the first person's card (the winning one) comes after it (second to last). If both players' cards **have the same values** - no one wins and the two cards must be removed from both hands. The **game is over** only when

one of the decks is left **without any cards**. You have to **display the result on the console** and **the sum of the remaining** cards: **"{First/Second} player wins! Sum: {sum}"**.

Examples

Input	Output
20 30 40 50 10 20 30 40	First player wins! Sum: 240
10 20 30 40 50 50 40 30 30 10	Second player wins! Sum: 50

7. Append Arrays

Create a program to **append several arrays** of numbers one after another.

- **Arrays** are **separated** by '|'
- Their **values** are **separated** by spaces (' ', one or several)
- Take all arrays starting from the **rightmost** and going to the **left** and place them in a new array in that order

Examples

Input	Output
1 2 3 |4 5 6 | 7 8	7 8 4 5 6 1 2 3
7 | 4 5|1 0| 2 5 |3	3 2 5 1 0 4 5 7
1| 4 5 6 7 | 8 9	8 9 4 5 6 7 1

8. *Anonymous Threat

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

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**.

Constrains

- 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

9. *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, which 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, which 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** of elements should be done in these cases, **also**. 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**.

Constrains

- 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**.

10. *SoftUni Course Planning

Help planning the next Programming Fundamentals course by keeping track of the lessons that will be included in the course, as well as all the exercises for the lessons. On the first input line, you will receive the initial schedule of lessons and exercises that are going to be part of the next course, separated by a comma and a space **", "**. Before the course starts, there are some changes to be made. 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, i**f 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

- 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.

Input	**Output**	**Comment**
`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, 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.