01. DHARMA Supplies

The periodic resupply drop is the name given to a DHARMA Initiative procedure in which new provisions and supplies are parachuted onto the Island from the DHARMA Logistics Warehouse.

Horace Goodspeed, the leader of the DHARMA Initiative on the Island, has asked you to help him **identify** and **collect** all parachuted supply crates.

You will start receiving **strings**, represinting different parts of the Island, until "**Collect**" command is received. The plane dropped many supply crates, all over the Island. Your task is to search in every part of the Island for the supplies, but be careful, the Hostiles poisoned some of the delivered crates.

A **supply crate** must start with an opening bracket "**[**", may have a **supply tag**(tag may be valid, may be not), may have a **supply body**, after the body must have the **exact same** supply tag and a closing bracket "**]**".

Example: "**[**{**supplyTag**}{**supplyBody**}{**supplyTag**}**]**"

There are two types of valid supply **tags**:

- **Food tag** - starts with "**#**" and has **N** digits(Ex: **#123**)
- **Drinks tag** - starts with "**#**" and has **N lowercase** english alphabet letters(Ex: **#asdf**)

N will be a positive integer calculated by the following formula:

- **number of ALL crates** / **number of Island parts**(inputlines).

N is representing the length of **all valid** supply tags. Any different length makes the tag invalid. Invalid tag means poisoned crate.

Supply bodies of the crates may contain english alphabet **letters** and/or **digits** and/or one or many **whitespaces**.

After all food and drink supply crates are collected, your task is simple. **Calculate** and **print** the **number of valid supply crates** collected, **amount of food** and **amount of drunks**. If **no** valid crates are found, print "**No supplies found!**".

- The **amount of food** in the food crates is calculated by **multiplying** the **sum** of the supply body **unique** ascci codes with the **length** of the supply tag(without the "**#**").
- The **amount of drinks** is calculated by **multiplying** the **sum** of the supply body ascci codes with the **sum** of the supply tag ascii codes(without the "**#**").

Input

- Until "**Collect**" is received, you must read input lines representing Island parts.

Output

- If you found some **valid** supplies, as output you must print **three** lines:
    1. **Number of valid supply crates** collected in format: "**Number of supply crates:** {number of supply crates}"
    2. **Amount of food** found in the crates: "**Amount of food collected:** {amount of food}"

3. **Amount of drinks** found in the crates: "**Amount of drinks collected:** {amount of drinks}"

- If no crates found, as output you must print a single line: "**No supplies found!**"

Constrains

- The **crates** will always start with "**[**" and end with "**]**".
- The **supply body** may contain english letters, numbers and one or many whitespaces.

Examples

Input	Output	Comment

`I <3 [#2017softuni#2017]!!!![#2beer#2]!!`
`machkai [#gRISHO#g]`
`Collect`
	`Number of supply crates: 2`
	`Amount of food collected: 313`
	`Amount of drinks collected: 40067`
		`Number of all crates: 3`
		`Number of lines: 2`
		`N: 3 / 2 = 1`
		
		`Valid crates: [#2beer#2] [#gRISHO#g]`
		
		`[#2beer#2] => (b + e + r) * 1 = 313`
		`[#gRISHO#g] = (R + I + S + H + O) * g = 40067`

`Bat Gi[##]orgi....`
`Zadyshawa[]m`
`Sa[#zzBasS#zz]`
`Collect`
	`No supplies found!`	`3 crates found, no valid ones.`

02. Crossfire

You will receive two integers which represent the dimensions of a matrix. Then, you must fill the matrix with increasing integers starting from 1, and continuing on every row, like this:
first row: 1, 2, 3… n
second row: n+1, n+2, n+3… n + n.

You will also receive several commands in the form of 3 integers separated by a space. Those 3 integers will represent a row in the matrix, a column and a radius. You must then **destroy** the cells which correspond to those arguments **cross-like.**

Destroying a cell means that, **that** cell becomes completely **nonexistent** in the matrix. Destroying cells **cross-like** means that you form a **cross figure**, with center point - equal to the cell with coordinates – the **given row** and **column**, and **lines** with length equal to the **given radius**. See the examples for more info.

The input ends when you receive the command “Nuke it from orbit”. When that happens, you must print what has remained from the initial matrix.

Input

- On the first line you will receive the dimensions of the matrix. You must then fill the matrix according to those dimensions.
- On the next several lines you will begin receiving 3 integers separated by a single **space**, which represent the row, col and radius. You must then destroy cells according to those coordinates.
- When you receive the command “Nuke it from orbit” the input ends.

Output

- The output is simple. You must print what is left from the matrix.
- Every row must be printed on a new line and every column of a row - separated by a space.

Constraints

- The dimensions of the matrix will be integers in the range [2, 100].
- The given rows and columns will be valid integers in the range [$-2^{31} + 1$, $2^{31} - 1$].
- The radius will be in range [0, $2^{31} - 1$].
- Allowed time/memory: 250ms/16MB.

03. Earthquake

You are given **N** – an integer. On the next **N** lines you will receive **sequences** of **integers**, which will represent seismic **activities**, and the seismic **waves** in them. The waves will hit the surface one by one, but they will also **neutralize** the **weaker** waves after them. The **integers** represent the waves’ **power**.

You must take the **first integer** from the **first activity** – that’s the **seismicity**. Then, you must start **comparing** it with the **other integers** from **the current activity**, by **order of input**. If the seismicity is **greater than** or **equal** to an **integer** (the wave), that integer must be **removed** from **the activity**. If the seismicity has **smaller** value than the integer, the **seismicity** should be **removed**.

When the current **seismicity** is **removed**, the **process** should **repeat** with the next **sequence** of **integers** in order. The **previous sequence** should go at the **back** of all **sequences** and should await its **next turn**.

When **all integers** from a **given activity** are **removed**, you should **remove** the **activity** itself.

Your task is to find how many seismic waves will hit the surface and their power are they. That means you must calculate how many **seismicities** you will find, before **all integers** and **activities** are **removed**.

Input

- On the first line of input you will receive **N**.
- On the next **N** lines you will receive **sequences** of **integers**, separated by a **space**.

Output

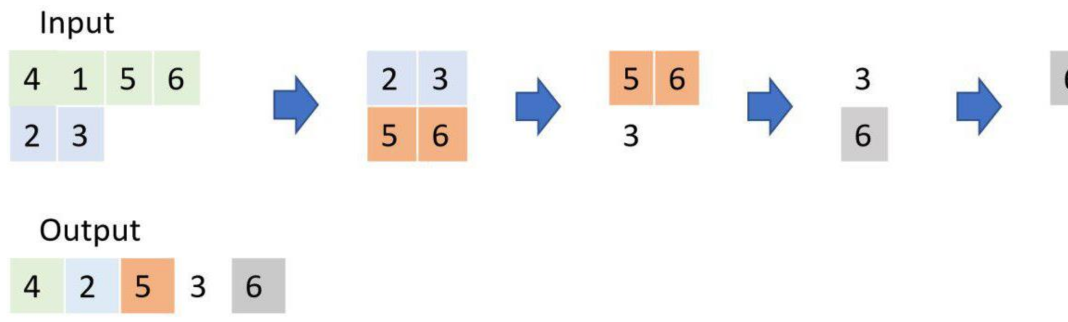
- As output you must print a single integer on the first line, indicating the amount of **seismicities** you've found.
- On the second line of output you must print the seismicities you found, by **order** of **entrance**.

Constrains

- Each of the integers in the input will be in the range [0, 1000].
- The sequences of integers will consist of [0, 300] integers.
- **All data** must be processed **by order of input**.
- Allowed time/memory: 100ms/16MB.

Examples

Input	Output	Comments

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

3
3 2 2 5
2 2 6
1 4
	6
	3 2 1 5 6 4	

Examples

Input	Output	Comment

5 5
3 3 2
4 3 2
Nuke it from orbit
	1 2 3 4 5
	6 7 8 10
	11 12 13
	16
	21
		Initial matrix:
		1 2 3 4 5
		6 7 8 9 10
		11 12 13 14 15
		16 17 18 19 20
		21 22 23 24 25
		Result from first destruction:
		1 2 3 4 5
		6 7 8 10
		11 12 13 15
		16
		21 22 23 25
		Result from second destruction:
		1 2 3 4 5
		6 7 8 10
		11 12 13
		16
		21

Input	Output

5 5
4 4 4
Nuke it from orbit
	1 2 3 4
	6 7 8 9
	11 12 13 14
	16 17 18 19

04. GUnit

GUnit is a unit testing framework – it provides the user with the opportunity to create unit tests which are very important to big project programming. It is still in beta though and you are given the job to format a database with all the classes, for each class - its methods, and of each of its methods – its unit tests.

You will be given several input lines which will contain info about a class's method's unit test. All valid lines will be in the following format:

{class name} | {method name} | {unit test name}

The elements are separated by a space, a '|' (vertical line) and another space. The valid class names, method names and unit test names can **only** contain English alphabet letters and digits, and **must** always start with a **capital letter**. All parameters must be at least **2** symbols long.

Any input line that does not follow the, specified above, format, should be **ignored**.

Your task is to save every unit test to its corresponding method and every method to its corresponding class in the database. If a class with the given name already exists you should add the new method with its test to it. If the method also exists in the given existing class, you should just add the new unit test to the corresponding method. If even the test is not new, you should do nothing.

There is also a specific way in which the classes, methods, and unit tests should be sorted. The classes should be ordered first by the amount of unit tests it has – descending, then by the amount of methods it has – ascending, and then alphabetically. The methods should be ordered by the amount of unit tests they have - descending, and then alphabetically. The unit tests should be ordered by the length of their names – ascending and then by alphabetically.

Input

- The input will come in the form of input lines.
- When you receive the command "It's testing time!" the input should stop and the output should start.

Output

- The output is simple. You must print all classes in the following format:
- "{class name}:
- "##{method1 name}:
- "####{test1 name}
- "####{test2 name}
- "##{method2 name}:
- "{class2 name}:
- …"
- For more info see the example below.

Constraints

- Every class will always have at least one method and every method will always have at least 1 test.
- Any input that does not consist only of what was specified as a valid format, is to be treated as invalid.
- All invalid input **must** be ignored.
- Allowed time/memory: 250ms/16MB

Examples

Input	**Output**

```
Rectangle | CalculateArea | Bla
Rectangle | CalculateArea | Bla2
Rectangle | CalculateArea | Bla3
Rectangle | CalculateArea | Bla4
Circles | CalculateArea | Bla
Circles | CalculateArea | Bla2
Circles | CalculateArea | Bla3
Circles | CalculateArea | Bla4
ASd | ASd | ASd
It's testing time!
```
	```
	Circles:
	##CalculateArea
	####Bla
	####Bla2
	####Bla3
	####Bla4
	Rectangle:
	##CalculateArea
	####Bla
	####Bla2
	####Bla3
	```

	####Bla4 ASd: ##ASd ####ASd