Lab: Stacks and Queues

You can check your solutions in [Judge]()

I. Working with Stacks

1. Reverse a String

Create a program that:

- **Reads** an **input string**
- **Reverses** it backwards (letter by letter, from the last to the first) **using a `Stack<T>`**
- **Prints** the result back at the console

Examples

Input	Output
`I Love C#`	`#C evoL I`
`Stacks and Queues`	`seueuQ dna skcatS`

Hints

- Use a **`Stack<string>`** and the methods **`Push()`**, **`Pop()`**.
- Push all chars from the input string, then pop and print them one by one.

2. Stack Sum

Create a program that:

- **Reads** an **input of integer numbers** and **adds** them to a **stack**.
- **Reads and executes commands** until **"end"** is received.
- Process the following commands:
  - **Add <n1> <n2>**: pushes two numbers into the stack
  - **Remove <n>**: removes the n elements from the stack or does nothing if the stack holds less than **n** elements.
- **Prints** the **sum** of the remaining elements of the **stack**.

Input

- On the **first line,** you will receive **an array of integers** (space-separated).
- On the **next lines**, until the "**end**" command is given, you will receive **commands** – a **single command** and **one** or **two** numbers after **the command, depending** on what **command** you are given.
  - If the **command** is "**add**", you will **always** be given **exactly two** numbers after the command, which you need to **add** to the **stack**.
  - If the **command** is "**remove**", you will **always** be given **exactly one** number after the command, which represents the **count** of the numbers you need to **remove** from the **stack. If** there are **not enough elements,** skip the command.
- Commands are **case-insensitive**, which means that "**Add**", "**add**" and "**aDD**" are the same command.
- A **single space** is used as a **separator** between commands and numbers.

Output

- When the **command** "**end**" is received, you need to **print the sum** of the **remaining** elements in the **stack**.

Examples

Input	Output	Comments

`1 2 3 4`
`adD 5 6`
`REmove 3`
`eNd`
	`Sum: 6`	The stack initially holds [1, 2, 3, 4].
		After the "Add 5 6" command, the stack holds [1, 2, 3, 4, 5, 6].
		After the "Remove 3" command, the stack holds [1, 2, 3].
		The sum of the elements [1, 2, 3] is 6.

`3 5 8 4 1 9`
`add 19 32`
`remove 10`
`add 89 22`
`remove 4`
	`Sum: 16`	The stack initially holds [3, 5, 8, 4, 1, 9].
		The stack now holds [3, 5, 8, 4, 1, 9, 19, 32].
		The command "Remove 10" is ignored (not enough elements).
		The stack now holds [3, 5, 8, 4, 1, 9, 19, 32, 89, 22].
		The stack now holds [3, 5, 8, 4, 1, 9].

`remove 3`
`end`
		The stack now holds [3, 5, 8].
		The sum of the elements [3, 5, 8] is 16.

Hints

- Use a **`Stack<int>`**
- Use the methods **`Push()`**, **`Pop()`**
- Commands **may** be given in **mixed case**.

3. Simple Calculator

Create a simple calculator that can **evaluate simple expressions** with only **addition** and **subtraction**. There will not be any parentheses. Numbers and operations are **space-separated**.

Solve the problem **using a `Stack`**.

Examples

Input	Output
2 + 5 + 10 - 2 - 1	14
2 - 2 + 5	5

Hints

- **Split** the input expression by space to **extract its tokens** (numbers and operations).
- **Reverse** the input tokens, then **push** them in a **Stack<string>**.
- Example:
  - Input expression: 2 + 5 + 10 - 2 - 1
  - Stack: 1 - 2 - 10 + 5 + 2
- **Pop** the last **number** (in the above example 2). It is the current result.
- **Pop** an **operation** and **number** (e. g. **+ 5**). Execute the operation. In our example: result = 2 + 5 = 7.
- **Repeat** the previous step until the stack gets empty.

4. Matching Brackets

We are given an arithmetic expression with brackets. Scan through the string and extract each sub-expression. Print the result back at the terminal.

Examples

Input	Output

1 + (2 - (2 + 3) * 4 / (3 + 1)) * 5	`(2 + 3)`
	`(3 + 1)`
	`(2 - (2 + 3) * 4 / (3 + 1))`

(2 + 3) - (2 + 3)	`(2 + 3)` `(2 + 3)`

Hints

- Scan through the expression from its start to its end, searching for brackets.
  - If you find an **opening** bracket, **push its index** (position in the input expression) into the stack.
  - If you find a **closing** bracket **pop the topmost** element from the stack. This is the **index** of the **opening bracket**.
  - Use the **current** and the popped index to extract the sub-expression.

II. Working with Queues

5. Print Even Numbers

Create a program that:

- **Reads** an array of **integers** and **adds** them to a **queue**.
- **Prints** the **even** numbers **separated** by "`, `".

Examples

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

11 13 18 95 2 112 81 46	`18, 2, 112, 46`

Hints

- Parse the input and enqueue all the numbers in a **`Queue<int>`**.
- **Dequeue** the elements one by one and print all **even** values.

6. Supermarket

You are given a **sequence of input strings**, each staying on a separate line. Each input string holds either a customer **name**, or the command "**Paid**" or the command "**End**". Your task is to read and process the input:

- When you receive a **customer name**, add it to the queue.
- When you receive the "**Paid**" command, **print** the customer names from the queue (each at separate line), then empty the queue.
- When you receive the "**End**" command, print the count of the remaining customers from the queue in the format: **`"{count} people remaining."`** and stop processing the commands (see the examples below).

Examples

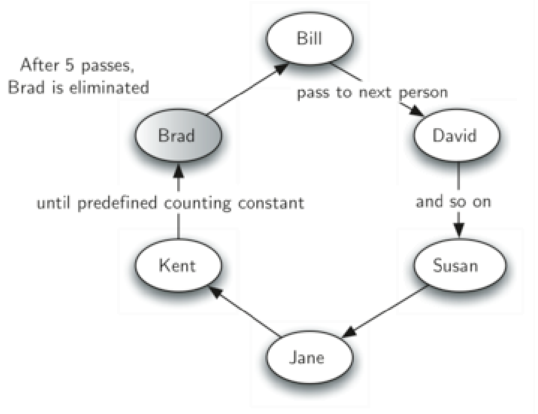
Input	Output
Liam Noah James **Paid** Oliver Lucas Logan Tiana **End**	Liam Noah James 4 people remaining.
Amelia Thomas Elias **End**	3 people remaining.

Hints

Use a queue and follow the description. Just read and implement the commands.

7. Hot Potato

Hot potato is a game in which **children form a circle and start passing a hot potato**. The counting starts with the first kid. **Every nth toss the child left with the potato leaves the game**. When a kid leaves the game, it passes the potato along to its next neighbor. This continues **until there is only one kid left**.



Create a program that simulates the game of Hot Potato. **Print every kid that is removed from the circle**. In the end, **print the kid that is left last**.

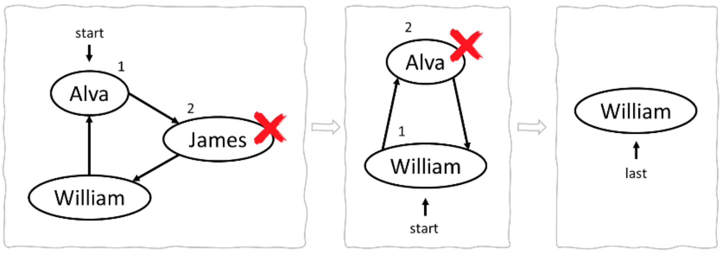
Examples

Input	Output		

`Alva James William`
`2`
	`Removed James`
	`Removed Alva`
	`Last is William`
		`Carter Dylan Jack Luke Gabriel`
		`1`
			`Removed Carter`
			`Removed Dylan`
			`Removed Jack`
			`Removed Luke`
			`Last is Gabriel`

`Lucas Jacob Noah Logan Ethan`
`10`
	`Removed Ethan`
	`Removed Jacob`
	`Removed Noah`
	`Removed Lucas`
	`Last is Logan`		

Illustration for the first example (Alva + James + William, n=2):



Hints

- Enqueue all kids in a **Queue<string>**.
- For each round do the following:
  - (n-1) times deque an element and enqueue it again.
  - Remove an element and print it (this is the nth element).
- Repeat the above until the queue remains holding only 1 element.

8. Traffic Jam

Create a program that simulates the **queue** that forms during a **traffic jam**. During a traffic jam, only **N** cars can **pass** the crossroads when the **light goes green**. Then the program reads the **vehicles** that **arrive** one by one and **adds** them to the **queue**. When the light **goes green N** number of cars **pass** the crossroads and **for each,** a **message** **"{car} passed!"** is displayed. When the "**end**" command is given, **terminate** the program and **display** a **message** with the **total number** of cars that **passed** the crossroads.

Input

- On the **first line,** you will receive **N** – the number of cars that can pass during a green light.
- On the **next lines,** until the "**end**" command is given, you will receive **commands** – a **single string**, either a **car** or "**green**".

Output

- Every time the "**green**" command is given, **print out** a message for **every car** that **passes** the crossroads in the format **"{car} passed!"**.
- When the "**end**" command is given, **print out** a message in the format **"{number of cars} cars passed the crossroads."**.

Examples

Input	Output		

`4`
`Hummer H2`
`Audi`
`Lada`
`Tesla`
`Renault`
`Trabant`
`Mercedes`
`MAN Truck`
`green`
`green`
`Tesla`
	`Hummer H2 passed!`
	`Audi passed!`
	`Lada passed!`
	`Tesla passed!`
	`Renault passed!`
	`Trabant passed!`
	`Mercedes passed!`
	`MAN Truck passed!`
	`8 cars passed the crossroads.`
		`3`
		`Enzo's car`
		`Jade's car`
		`Mercedes CLS`
		`Audi`
		`green`
		`BMW X5`
		`green`
		`end`
			`Enzo's car passed!`
			`Jade's car passed!`
			`Mercedes CLS passed!`
			`Audi passed!`
			`BMW X5 passed!`
			`5 cars passed the crossroads.`

Renault Trabant end			

Exercise: Stacks and Queues

- You can check your solutions in [Judge]
- Ask your questions here https://www.slido.com/ by entering the code **`#csharp-advanced`**

1. Basic Stack Operations

Play around with a stack. You will be given an integer **N** representing the number of elements to push into the stack, an integer **S** representing the number of elements to pop from the stack, and finally an integer **X**, an element that you should look for in the stack. If it's found, print "**`true`**" on the console. If it isn't, print the **smallest** element currently present in the stack. If there are **no elements** in the sequence, print **`0`** on the console.

Input

- On the first line, you will be given **N**, **S** and **X,** separated by a single space.
- On the next line, you will be given **N** number of integers.

Output

- On a single line, print either **true** if **X** is present in the stack, otherwise print the **smallest** element in the stack. If the stack is **empty**, print **`0`.**

Examples

Input	Output	Comments

`5 2 13`
`1 13 45 32 4`
	`true`	We have to **push 5** elements. Then we **pop 2** of them. Finally, we have to check whether 13 is present in the stack. Since it is, we print **`true`**.

`4 1 666`
`420 69 13 666`
	`13`	We have to **push 4** elements and then **pop 1** one of them. Then, we have to check whether **666 is present** in the stack. Since **it isn't**, we have to print the **smallest** element of the stack, which is **13**.

`3 3 90`
`90 90 90`
	`0`	We have to **push 3** elements and then **pop** them. Since there are **no elements** left in the stack, we print **0** at the end.

2. Basic Queue Operations

Play around with a queue. You will be given an integer **N** representing the number of elements to enqueue (**add**), an integer **S**, representing the **number of elements** to **dequeue** (**remove**) from the queue, and finally an integer **X**, an element that you should look for in the **queue**. If it is, print **true** on the console. If it's not printed the **smallest element** is currently present in the queue. If there are **no elements** in the sequence, print **`0`** on the console.

Examples

Input	Output	Comments

`5 2 32`
`1 13 45 32 4`
	`true`	We have to **enqueue 5** elements. Then we **dequeue 2** of them. Finally, we have to check whether 32 is present in the queue. Since it is. we print **`true`**.

`4 1 666`
`666 69 13 420`
	`13`	We have to **enqueue 4** elements and then **dequeue 1** one of them. Then, we have to check whether **666 is present** in the queue. Since **it isn't**, we have to print the **smallest** element of the queue, which is **13**.

`3 3 90`
`90 0 90`
	`0`	We have to **enqueue 3** elements and then **dequeue** them. Since there are **no elements** left in the queue, we print **0** at the end.

3. Maximum and Minimum Element

You have an empty sequence and you will be given **N** queries. Each query is one of these three types:

1 x – **Push** the element x into the stack.

2 – **Delete** the element present at the **top** of the **stack**.

3 – **Print** the **maximum** element in the stack.

4 – **Print** the **minimum** element in the stack.

After you go through all of the queries, print the stack in the following format:
`"{n}, {n₁}, {n₂} …, {nₙ}"`

Input

- The first line of input contains an integer – **N**.
- The next **N** lines each contain an above-mentioned query. (**It is guaranteed that each query is valid.**)

Output

- For each type 3 or 4 queries, print the **maximum**/minimum element in the stack on a new line.

Constraints

- 1 ≤ N ≤ 105
- 1 ≤ x ≤ 109
- 1 ≤ type ≤ 4
- If there are **no elements** in the stack, **don't print anything** on commands 3 and 4

Examples

Input	**Output**		
9 1 97 2 1 20 2 1 26 1 20 3 1 91 4	26 20 91, 20, 26	9 1 47 1 66 1 32 4 3 1 25 1 16 1 8 4	32 66 8 8, 16, 25, 32, 66, 47

4. Fast Food

You have a fast-food restaurant and most of the food that you're offering is previously prepared. You need to know if you will have enough food to serve lunch to all of your customers. Write a program that checks the orders' quantity. You also want to know the client with the **biggest** order for the day, because you want to give him a discount the next time he comes.

First, you will be given the **quantity of the food** that you have for the day (an integer number). Next, you will be given **a sequence of integers**, each representing the **quantity of order**. Keep the orders in a **queue**. Find the **biggest order** and **print** it. You will begin servicing your clients from the **first one** that came. Before each order, **check** if you have enough food left to complete it. If you have, **remove the order** from the queue and **reduce** the amount of food you have. If you succeeded in servicing all of your clients, print:
`"Orders complete".`
If not – print:
`"Orders left: {order1} {order2} .... {orderN}"`.

Input

- On the first line, you will be given the quantity of your food – **an integer** in the range [0…1000].
- On the second line, you will receive a sequence of integers, representing each order, **separated by a single space.**

Output

- Print the quantity of the biggest order.
- Print "`Orders complete`", if the orders are complete.
- If there are orders left, print them in the format given above

Constraints

- The input will always be valid.

Examples

Input	Output

`348`
`20 54 30 16 7 9`
	`54`
	`Orders complete`

`499`
`57 45 62 70 33 90 88 76`
	`90`
	`Orders left: 76`

5. Fashion Boutique

You own a fashion boutique and you receive a delivery once a month in a huge box, which is full of clothes. You have to arrange them in your store, so you take the box and start **from the last piece** of clothing on the top of the pile **to the first one** at the bottom. Use a **stack** for this purpose. Each piece of clothing has its **value** (an integer). You have to **sum** their values, while you take them out of the box. You will be given an integer, representing the **capacity** of a rack. While the sum of the clothes is **less** than the capacity, **keep summing** them. If the sum becomes **equal** to the capacity, you have to **take a new rack** for the **next clothes** if there are **any left** in the box. If it becomes **greater** than the capacity**, don't add** the piece of clothing to the current rack and take a new one. In the end, print **how many racks** you have used to hang all of the clothes.

Input

- On the first line, you will be given **a sequence of integers**, representing the clothes in the box, separated **by a single space.**
- On the second line, you will be given **an integer**, representing the capacity of a rack.

Output

1. Print the **number of racks**, needed to hang all of the clothes from the box.

Constraints

2. The values of the clothes will be integers in the range [0…20].
3. There will never be more than 50 clothes in a box.
4. The capacity will be an integer in the range [0…20]
5. **None** of the integers from the box will be **greater** than than the **value** of the **capacity.**

Examples

Input	Output		

`5 4 8 6 3 8 7 7 9`
`16`
	`5`	`1 7 8 2 5`
		`4 7 8 9 6`
		`3 2 5 4 6`
		`20`
			`5`

6. Songs Queue

Write a program that keeps track of a song's queue. The **first** song that is put in the queue, should be the **first** that **gets played**. A song cannot be added, if it is currently in the queue.

You will be given **a sequence of songs**, separated by a comma and a single space. After that, you will be given **commands until** there are **no songs enqueued**. When there are **no more songs** in the queue **print** "`No more songs!`" and **stop** the **program**.

The possible commands are:

- **"`Play`" - plays a song (removes it from the queue)**
- **"`Add {song}`" - adds the song to the queue, if it isn't contained already, otherwise print "`{song} is already contained!`"**
- **"`Show`" - prints all songs in the queue, separated by a comma and a white space (start from the first song in the queue to the last)**

Input

- On the first line, you will be given a sequence of strings, separated by a comma and a white space.
- On the next lines, you will be given commands until there are no songs in the queue.

Output

- While receiving the commands, print the proper messages described above
- After the command "**Show**", print the songs from the **first** to the **last.**

Constraints

- The input **will always be valid** and in the **formats** described above.
- There **might** be commands **even after** there are **no songs in the queue** (ignore them).
- There will never be duplicate songs in the initial queue.

Examples

Input	Output

`All Over Again, Watch Me`
`Play`
`Add Watch Me`
`Add Love Me Harder`
`Add Promises`
`Show`
`Play`
`Play`
`Play`
`Play`
	`Watch Me is already contained!`
	`Watch Me, Love Me Harder, Promises`
	`No more songs!`

7. Truck Tour

Let's suppose there is a circular route for lorries. There are **N** petrol pumps on that toute. Petrol pumps are numbered `0` to (N−1) (both inclusive). You will receive **information (array)**, corresponding to each of the petrol pumps: [0] the **amount of petrol (in litres)** that particular petrol pump will give, and [1] the **distance(in kilometers) from the current petrol pump** to the next petrol pump.
You have a tank of infinite capacity carrying no petrol. Your task is to figure out, which is the first possible petrol pump, from which the lorry will be able to complete the route. Consider that the lorry will stop at **each of the petrol pumps**. The lorry will move one kilometer for each liter of petrol.

Input

- The first line will contain the value of **N**
- The next **N** lines will contain a pair of integers each, i.e. the amount of petrol that petrol pump will give and the distance between that petrol pump and the next petrol pump.

Output

- An integer which will be the smallest index of the petrol pump from which we can start the tour.

Constraints

- 1 ≤ N ≤ 1000001
- 1 ≤ Amount of petrol, Distance ≤ 1000000000

Examples

Input	Output

`3`
`1 5`
`10 3`
`3 4`
	`1`

8. Balanced Parentheses

Given a sequence consisting of parentheses, determine whether the expression is **balanced**. A sequence of parentheses is balanced, if every **open parenthesis** can be **paired uniquely** with a **closing parenthesis** that occurs **after** the former. Also, the **interval between** them **must** be **balanced**. You will be given **three** types of parentheses: (, {, and [.
`{[()]}` - This is a balanced parenthesis.
`{[(])}` - This is not a balanced parenthesis.

Input

- Each input consists of a single line, **the sequence of parentheses**.

Output

- For each test case, print on a new line "**YES**", if the parentheses are balanced. Otherwise, print "**NO**". Do not print the quotes.

Constraints

- 1 ≤ len_s ≤ 1000, where the len_s is the length of the sequence.
- Each character of the sequence **will be one of** `{`, `}`, `(`, `)`, `[`, `]`.

Examples

Input	Output
`{[()]}`	YES
`{[(])}`	NO
`{{[[(())]]}}`	YES

9. Simple Text Editor

You are given an empty text. Your task is to implement 4 commands related to manipulating the text

- `1 someString` - **appends** someString to the end of the text.
- `2 count` - **erases** the last **`count`** elements from the text.
- `3 index` - **returns** the element at position **index** from the text.
- 4 - **undoes** the last not undone command of type *1* or *2* and returns the text to the state before that operation.

Input

- The first line contains **n** – the number of operations.
- Each of the following **n** lines contains the name of the operation, followed by the command argument, if any, separated by space in the following format "**`CommandName Argument`**".

Output

- For each operation of type ***3*** print a single line with the returned character of that operation.

Constraints

- 1 ≤ N ≤ 105.
- The length of the text will not exceed 1000000.
- All input characters are English letters.
- It is guaranteed that the sequence of input operations is possible to perform.

Examples

Input	Output	Comments
8 1 abc 3 3 2 3 1 xy 3 2 4 4 3 1	c y a	There are 8 operations. Initially, the text is empty. In the first operation, we append **abc** to the text. Then, we print its 3rd character, which is **c** at this point. Next, we erase its last 3 characters, **abc**. After that, we append **xy** to the text. The text becomes **xy** after these previous two modifications. Then, we are asked to return the 2nd character of the text, which is **y**. After that, we have to undo the last update to the text, so it becomes empty. The next operation asks us to undo the update before that, so the text becomes **abc** again. Finally, we are asked to print its 1st character, which is **a** at this point.
9 1 HelloThere 3 7 2 2	h o h P	There are 9 operations. Initially, the text is empty. The first operation appends **HelloThere** to the text. Then, the second operation returns the 7thcharacter, which is **h** at this point. Next, we have to erase the last 2 characters from the text, which are "**re**". After that, we print the 5th character of the text – 'h'.

`3 5`
`4`
`3 7`
`4`
`1`
`TestPassed`
`3 5`
		Then, we are asked to undo last update of the text, so we have to append "**`re`**" to the end of the text.
		Again, we have to return the 7th element of the text, which is '**h**'.
		After that, we have to undo the last undone operation of type 1 or 2. The last **undone** operation is `1 HelloThere`, so we remove **`HelloThere`** from the text.
		The next operation appends **`TestPassed`** to the end of the text.
		Finally, we return the 5th element from the text – 'P'.

10. *Crossroads

Our favorite super-spy action hero Sam is back from his mission from the previous exam and he has finally found some time to go on a **holiday**. He is taking his wife somewhere nice and they're going to have a really good time, but first, they have to get there. Even on his holiday trip, Sam is still going to run into some **problems** and the first one is, of course, getting to the airport. Right now, he is stuck in a traffic jam at a **very active crossroad** where a lot of **accidents** happen.

Your job is to keep track of traffic at the crossroads and report whether a **crash happened** or everyone **passed** the **crossroads safely** and our hero is one step closer to a much-desired vacation.

The road Sam is on has a **single lane** where cars queue up until the **light goes green**. When it does, they start passing one by one during the **green light** and the **free window** before the **intersecting road's light** goes **green**. During **one second** only **one part** of a **car** (a **single character**) passes the crossroads. If a car is still at the crossroads when the **free window** ends, it will get hit at the **first character** that is still in the crossroads.

Input

- On the **first line,** you will receive the duration of the **green light** in seconds – an **integer in the range [1…100].**
- On the **second line,** you will receive the duration of the **free window** in seconds – an **integer in the range [0…100].**
- On the **following lines**, until you receive the "**END**" command, you will receive one of two things:
    - A **car** – a **string** containing any ASCII character, or
    - The command "**`green`**" indicates the **start** of a **green light cycle**

A **green light cycle** goes as follows:

- During the **green light,** cars will enter and exit the crossroads one by one.
- During the **free window,** cars will only exit the crossroads.

Output

- If a **crash happens**, **end the program** and print:

`"A crash happened!"`
`"{car} was hit at {characterHit}."`

- If everything **goes smoothly** and you receive an "**END**" command, print:

`"Everyone is safe.".`
`"{totalCarsPassed} total cars passed the crossroads.".`

Constraints

- The input will be **within the constraints** specified above and will **always be valid**. There is **no need** to check it explicitly.

Examples

Input	Output	Comments

`10`
`5`
`Mercedes`
`green`
	`Everyone is safe.`
	`3 total cars passed the crossroads.`
		During the first green light (**10** seconds), the **`Mercedes`** (**8**) passes safely.
		During the second green light, the **`Mercedes`** (**8**) passes safely and there are **2 seconds left**.

`Mercedes`
`BMW`
`Skoda`
`green`
`END`
		The **`BMW` enters** the crossroads and when the green light ends, it still has **1 part** inside ('**`W`**') but has **5 seconds** to leave and passes successfully.
		The **`Skoda` never enters** the crossroads, so **3 cars passed successfully**.

`9`
`3`
`Mercedes`
`Hummer`
`green`
`Hummer`
`Mercedes`
`green`
`END`
	`A crash happened!`
	`Hummer was hit at e.`
		`Mercedes` (**8**) passes successfully and **`Hummer`** (**6**) enters the crossroads but only the '**`H`**' passes during the green light. There are **3** seconds of a free window, so "**`umm`**" passes and the **`Hummer`** gets hit at '**`e`**' and the program ends with a **crash**.

11. *Key Revolver

Our favorite super-spy action hero Sam is back from his mission in another exam, and this time he has an even more difficult task. He needs to **unlock a safe**. The problem is that the safe is **locked** by **several locks in a row**, which all have **varying sizes**.
Our hero possesses a special weapon though, called the **Key Revolver**, with special bullets. Each **bullet** can unlock a **lock** with a **size equal to or larger than** the **size** of the **bullet**. The bullet goes into the keyhole, then explodes, **destroying** it. Sam **doesn't know the size** of the locks, so he needs to just shoot at all of them until the safe run out of locks.
What's behind the safe, you ask? Well, intelligence! It is told that Sam's sworn enemy – **Nikoladze**, keeps his **top-secret Georgian Chacha Brandy** recipe inside. It's valued differently across different times of the year, so Sam's boss will tell him what it's worth over the radio. One last thing, every bullet Sam fires will also cost him money, w**hich will be deducted from his pay** from the price of the intelligence.
Good luck, operative.

Input

- On the **first line** of input, you will receive the price of each **bullet** – an **integer in the range [0…100].**
- On the **second line,** you will receive the **size of the gun barrel** – an **integer in the range [1…5000].**
- On the **third line,** you will receive the **bullets** – a **space-separated integer sequence** with **[1…100] integers.**
- On the **fourth line,** you will receive the **locks** – a **space-separated integer sequence** with **[1…100] integers.**
- On the **fifth line,** you will receive the **value of the intelligence** – an **integer in the range [1…100000].**

After Sam receives all of his information and gear (**input**), he starts to **shoot the locks front-to-back**, while going through the bullets **back-to-front**.
If the **bullet** has a **smaller or equal** size to the **current lock**, print "**`Bang!`**", then **remove the lock**. If not, print **"`Ping!`"**, leaving the lock **intact**. The bullet is removed in **both cases**.
If Sam runs out of bullets in his barrel, print **"`Reloading!`"** in the console, then continue shooting. If there aren't any bullets left, **don't** print it.
The program ends when Sam **either runs out of bullets** or the safe **runs out of locks**.

Output

- If Sam **runs out of bullets** before the safe runs out of **locks**, print **`"Couldn't get through. Locks left: {locksLeft}"`**.

- If Sam manages to **open the safe**, print **`"{bulletsLeft} bullets left. Earned ${moneyEarned}"`.**

Make sure to account for the **price of the bullets** when calculating the **money earned**.

Constraints

- The input will be **within the constraints** specified above and will **always be valid**. There is **no need** to check it explicitly.
- There will **never** be a case where Sam breaks the lock and ends up with a **negative balance**.

Examples

Input	Output	Comments

50
2
11 10 5 11 10 20
15 13 16
1500
	Ping!
	Bang!
	Reloading!
	Bang!
	Bang!
	Reloading!
	2 bullets left. Earned $1300
		20 shoots lock **15** (**ping**)
		10 shoots lock **15** (**bang**)
		11 shoots lock **13** (**bang**)
		 5 shoots lock **16** (**bang**)
		
		Bullet cost: 4 * 50 = $200
		Earned: 1500 – 200 = $1300

20
6
14 13 12 11 10 5
13 3 11 10
800
	Bang!
	Ping!
	Ping!
	Ping!
	Ping!
	Ping!
	Couldn't get through. Locks left: 3
		5 shoots lock **13** (**bang**)
		10 shoots lock **3** (**ping**)
		11 shoots lock **3** (**ping**)
		12 shoots lock **3** (**ping**)
		13 shoots lock **3** (**ping**)
		14 shoots lock **3** (**ping**)

33
1
12 11 10
10 20 30
100
	Bang!
	Reloading!
	Bang!
	Reloading!
	Bang!
	0 bullets left. Earned $1
		10 shoots lock **10** (**bang**)
		11 shoots lock **20** (**bang**)
		12 shoots lock **30** (**bang**)
		
		Bullet cost: 3 * 33 = $99
		Earned: 100 – 99 = $1

12. *Cups and Bottles

You will be given a **sequence of integers** – each indicating a **cup's capacity**. After that, you will be given **another sequence of integers** – a **bottle with water** in it. Your job is to try to **fill up** all of the cups.

The filling is done by picking **exactly one** bottle at a time. You must start picking from **the last received bottle** and start filling from **the first entered cup**. If the current bottle has **N** water, you **give** the **first entered cup N** water and **reduce** its integer value by **N**.

When a cup's **integer value** reaches **0 or less**, it **gets removed**. The current cup's value may be **greater** than the current bottle's value. **In that case,** you **pick bottles until** you reduce the cup's integer value to **0 or less**. If a bottle's value is **greater or equal to** the cup's **current** value, you fill up the cup, and **the remaining water becomes wasted**. You should **keep track of the wasted litter of water** and **print it at the end of the program**.

If you **have managed** to **fill up all of the cups**, print the **remaining water bottles**, from the **last entered – to the first**, otherwise you must print the **remaining cups**, by **order of entrance** – from the **first entered – to the last**.

Input

- On the **first line** of input, you will receive the integers, representing the **cups' capacity**, **separated** by a **single space**.
- On the **second line** of input, you will receive the integers, representing the **filled bottles**, **separated** by a **single space**.

Output

- On the first line of output, you must print the remaining bottles or the remaining cups, depending on the case you are in. Just **keep** the **orders of printing exactly as specified**.
  - `"Bottles: {remainingBottles}"` or `"Cups: {remainingCups}"`
- On the second line, print the wasted litters of water in the following format: `"Wasted litters of water: {wastedLittersOfWater}"`.

Constraints

- All of the given numbers will be valid integers in the range [1…500].
- It is safe to assume that there will be **NO** case in which the water is **exactly as much** as the cups' values so that at the end there are no cups and no water in the bottles.
- Allowed time/memory: 100ms/16MB.

Examples

Input	Output	Comment

`4 2 10 5`
`3 15 15 11 6`
	`Bottles: 3`
	`Wasted litters of water: 26`
		We take the first entered cup and the last entered bottle, as it is described in the condition.
		6 – 4 = 2 – we have 2 more so the wasted water becomes 2.
		11 – 2 = 9 – again, it is more, so we add it to the previous amount, which is 2 and it becomes 11.
		15 – 10 = 5 – wasted water becomes 16.
		15 – 5 = 10 – wasted water becomes 26.
		We've managed to fill up all of the cups, so we print the remaining bottles and the total amount of wasted water.

`1 5 28 1 4`
`3 18 1 9 30 4 5`
	`Cups: 4`
	`Wasted litters of water: 35`	

`10 20 30 40 50`
`20 11`
	`Cups: 30 40 50`
	`Wasted litters of water: 1`	