**Exercise: Lists as Stacks and Queues**

Problems for exercise and homework for the [Python Advanced Course @SoftUni](https://softuni.bg/courses/python-advanced).

Submit your solutions in the SoftUni judge system at <https://judge.softuni.bg/Contests/1831>.

## Reverse Numbers with a Stack

Write a program that reads from the console a string with **N integers, separated by a single space,** and **reverses them using a stack. Print** the reversed integers on **one line, separated by a single space**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 2 3 4 5 | 5 4 3 2 1 |
| 1 | 1 |

## 2. Maximum and Minimum Element

You have an empty sequence. You will receive an integer – **N***.* On the next **N** lines you will receive queries. Each query is one of these four types:

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

**'2'** – **Delete** the element 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 the queries, print the **stack from the top to the bottom** in the following format:

"{n}, {n1}, {n2} …, {nn}"

### Constraints

* **It is guaranteed that each query is valid**
* **1 ≤ N ≤ 105**
* **1 ≤ x ≤ 109**
* **1 ≤ type ≤ 4**

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 9  1 97  2  1 20  2  1 26  1 20  3  1 91  4 | 26  20  91, 20, 26 |
| 10  2  1 47  1 66  1 32  4  3  1 25  1 16  1 8  4 | 32  66  8  8, 16, 25, 32, 66, 47 |

## 3. 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 your customers.

Write a program which checks the orders' quantity. You also want to know who the client with the biggest order for that day is, because you want to give him a discount.

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 an 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 your clients, print: **"Orders complete"**. Otherwise, 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 biggest order
* Print "**Orders complete**" if the orders are completed
* 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 |

## 4. Fashion Boutique

You own a fashion boutique, and you receive a delivery of a **huge box** **of clothes**, represented as a **sequence of integers**. On the next line, you will be given an integer representing the **capacity** for **one rack** in your store.

You must arrange the clothes in the store, and you use the racks to hang up every piece of clothing. You start **from the last piece** of clothing on the top of the pile **to the first one** at the bottom. Use a **stack** for the purpose. Each piece of clothing has its **value** (an integer). You must **sum** their values, while you take them out of the box. If the sum becomes **equal** to the capacity of the current rack you must **take a new one** for the **next clothes**, if there are **any left** in the box. If the sum becomes **greater** than the capacity, **do not hang** the piece of clothing to the current rack. Take a new rack and then hang up the piece of clothing. In the end, print **how many racks** you have used to hang up 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

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

### Constraints

* The values of the clothes will be integers in the range [0,20]
* There will never be more than **50** clothes in a box
* The capacity will be an integer in the range [0,20]
* **None** of the integers from the box will be **greater** than then 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 |

## 5. Truck Tour

On a circle road there are **N** petrol pumps. Petrol pumps are numbered **0** to (**N−1**) (both inclusive). For each petrol pump you will receive **two pieces of information**:

* the **amount of petrol** that petrol pump will give
* the **distance from that petrol pump** to the next petrol pump (kilometers)

Initially, you have a tank of infinite capacity carrying no petrol. You can start the tour at **any** of the petrol pumps. Calculate the **first point** from where the truck will be able to **complete the circle**. Consider that the truck will stop at **each of the petrol pumps**. The truck will move one kilometer for each liter of the petrol.

### Input

* On the first line you will receive the **N**-numberpetrol pumps
* On the next **N**-lines you will receive the amount of petrol that petrol pump will give and the distance between that petrol pump and the next petrol pump, separated by single space

### Output

* An integer which will be **the smallest index** of the petrol pump from which you 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 |

## 6. Balanced Parentheses

You will be given a sequence consisting of parentheses. Your job is to determine whether the expression is **balanced**. A sequence of parentheses is balanced if every **opening** **parenthesis** has a corresponding **closing** **parenthesis** that occurs **after** the former. There will be **no** **interval symbols** **between** the parentheses. You will be given **three** types of parentheses: (, {, and [.

**{[()]} - Parentheses are balanced.**

**(){}[] - Parentheses are balanced.**

**{[(])} - Parentheses are NOT balanced.**

### Input

* On a **single line** you will receive a **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 ≤ lens ≤ 1000**, where lens is the length of the sequence.
* Each character of the sequence **will be one of** {, }, (, ), [, ].

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| {[()]} | YES |
| {[(])} | NO |
| {{[[(())]]}} | YES |

## \*Robotics

Somewhere in the future, there is a robotics factory. The current project is assembly line robots.

Each robot has a **processing time –** it is the **time in seconds** the robot needs to process a product. When a **robot is free** it should **take a product for processing** and **log his name, product and processing start time**.

Each robot **processes a product coming from the assembly line**. A **product is coming** from the line **each second** (so the first product should appear at [start time + 1 second]). If a product passes the line and **there is not a free robot** to take it, it should be **queued at the end of the line again**.

The robots are **standing on the line in the order of their appearance**.

### Input

* On the first line, you will receive the names of the robots and their processing times in the format **"robotName-processTime;robotName-processTime;robotName-processTime..."**
* On the second line, you will get the starting time in format **"hh:mm:ss"**
* Next, until the **"End"** command, you will get a product on each line.

### Output

* Every time a **robot takes a product,** you should print: **"{robotName} - {product} [hh:mm:ss]"**

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| ROB-15;SS2-10;NX8000-3  8:00:00  detail  glass  wood  apple  End | ROB - detail [08:00:01]  SS2 - glass [08:00:02]  NX8000 - wood [08:00:03]  NX8000 - apple [08:00:06] |
| ROB-60  7:59:59  detail  glass  wood  sock  End | ROB - detail [08:00:00]  ROB - sock [08:01:00]  ROB - wood [08:02:00]  ROB - glass [08:03:00] |

## \*Crossroads

Our favorite super-spy action hero Sam is back from his mission in 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 getting to the airport. Right now, he is stuck in a traffic jam at a **crossroads** where a lot of **accidents** happen.

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

Sam is on a **single** **lane of cars** which queue up until the **light** **goes** **green**. When it does, they start passing one by one on a flashing **green light** and during the **free window** before the **intersecting** **road's** **light** goes **green**. For each **second** only **one** **part** of a **car** (a **single** **character**) passes the crossroad. If a car is **still in the middle of 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** **[1 … 100]**
* On the **second line**, you will receive the duration of the **free** **window** in seconds – an **integer** **[0 … 100]**
* On the **following lines**, until you receive the "**END**" command, you will receive one of two things:
  + A **car** – a **string** containing the model of the car, or
  + The command "**green**" which 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 {character\_hit}."
* If everything **goes** **smoothly** and you receive an "**END**" command, print:  
  "Everyone is safe."  
  **"**{total\_cars\_passed} total cars passed the crossroads.**"**

### Constraints

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

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 10  5  Mercedes  green  Mercedes  BMW  Skoda  green  END | 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.  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 free window, so "umm" passes and the Hummer gets hit at 'e' and the program ends with a crash. |

## 9. \*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 posesses 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, completely **destroying** it. Sam **doesn't know the size** of the locks, so he needs to just shoot at all of them, until the safe runs 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, **which 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!" on 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: {locks\_left}"
* If Sam manages to **open the safe**, print:  
  "{bullets\_left} bullets left. Earned ${money\_earned}"

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

### Constraints

* The input will be **within the constaints** 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 а **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 |

## 10. \*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 the cups.

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**. It is **possible** that the current cup's value is **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 litters of water** and **print it at the end of the program**.

If you **have managed** to **fill up all 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: {remaining\_bottles}"** or **"Cups: {remaining\_cups}"**
* On the second line print the wasted litters of water in the following format: **"Wasted litters of water: {wastedLitters\_of\_water}."**

### Constraints

* All the given numbers will be valid integers in the range **[1, 1000]**.
* 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.

### 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 |  |