# Stacks and Queues

## Reverse Strings

Create a program that:

* **Reads** an **input string**
* **Reverses** it **using a** Stack<T>
* **Prints** the result back at the terminal

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| I Love C# | #C evoL I |
| Stacks and Queues | seueuQ dna skcatS |

## Stack Sum

Create a program that:

* **Reads** an **input of integer numbers** and **adds** them to a **stack**
* **Reads command** until **"end"** is received
* **Prints** the **sum** of the remaining elements of the **stack**

### Input

* On the **first** **line,** you will receive **an array of integers**
* 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** receive **exactly two** numbers after the command which you need to **add** to the **stack**
* If the **command** is "**remove**", you will **always** receive **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.

### Output

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

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 2 3 4  adD 5 6  REmove 3  eNd | Sum: 6 |
| 3 5 8 4 1 9  add 19 32  remove 10  add 89 22  remove 4  remove 3  end | Sum: 16 |

## Simple Calculator

**Create a simple calculator** that can **evaluate simple expressions** with only addition and subtraction. There will not be any parentheses.

Solve the problem **using a Stack**.

### Examples

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

## 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) |

## 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 |

## 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. 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 |
| Lucas Jacob Noah Logan Ethan  10 | Removed Ethan  Removed Jacob  Removed Noah  Removed Lucas  Last is Logan |
| Carter Dylan Jack Luke Gabriel  1 | Removed Carter  Removed Dylan  Removed Jack  Removed Luke  Last is Gabriel |

## 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  Renault  Trabant  end | 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. |

## 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 |  |

## 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 |
| 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 |

## 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 |

## 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

* Print the **number of racks**, needed to hang all of 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 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 |

## 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! |

## Truck Tour

Suppose there is a circle. There are **N** petrol pumps on that circle. Petrol pumps are numbered 0 to (N−1) (both inclusive). You have **two pieces of information** corresponding to each of the petrol pumps: (1) the **amount of petrol** that particular petrol pump will give, and (2) the **distance from that petrol pump** to the next petrol pump.

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 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 |

## 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 ≤ lens ≤ 1000, where the lens is the length of the sequence.
* Each character of the sequence **will be one of** {, }, (, ), [, ].

### Examples

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