Exercises: Stacks and Queues

This document defines the exercises for the "Java Advanced" course @ Software University. Please submit your solutions (source code) to all below-described problems in Judge.

1. Reverse Numbers with a Stack

Write a program that reads **N integers** from the console and **reverses them using a stack**. Use the **`ArrayDeque<Integer>`** class. Just put the input numbers in the stack and pop them.

Examples

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

2. Basic Stack Operations

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 an integer **X**, an element **that you should check whether is present in the stack**. If it is, print "**`true`**" on the console. If it's not, print the smallest element currently present in the stack.

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 a line of numbers **separated by one or more white spaces**.

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	Pop one element (666) and then check if 666 is present in the stack. It's not, so print the smallest element (13).
3 3 90 90 90 90	0	

3. Maximum Element

You have an empty sequence, and you will be given **N** commands. Each command is one of the following 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.

Input

- The first line of input contains an integer **N**, where **1 ≤ N ≤ 10^5**.
- The next **N** lines contain commands. All commands will be valid and in the format described.
- The element **X** will be in the range **1 ≤ X ≤ 10^9**.
- The **type of the command** will be in the range **1 ≤ Type ≤ 3.**

Output

- For each command of **type** "**3**", **print the maximum element** in the stack on a new line.

Examples

Input	Output	Comments
9 1 97 2 1 20 2 1 26 1 20 3 1 91 3	26 91	9 commands Push 97 Pop an element Push 20 Pop an element Push 26 Push 20 Print the maximum element (26) Push 91 Print the maximum element (91)
7 1 81 2 1 14 2 1 14 1 47 3	47	

4. Basic Queue Operations

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/poll) from the queue, and finally, an integer **X**, an element that you should **check whether is present in the queue**. If it is - prints **true** on the console, if it is not - **print the smallest element currently present in the queue**.

Examples

Input	Output	Comments

5 2 32 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 1 666 666 69 13 420	13	
3 3 90 90 90 90	0	

5. 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.
- **1 ≤ Length of sequence ≤ 1000.**
- Each character of the sequence will be one of the following: **{**, **}**, **(**, **)**, **[**, **]**.

Output

- For each test case, print on a new line "**YES**" if the parentheses are balanced. Otherwise, print "**NO**".

Examples

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

6. Recursive Fibonacci

Each member of the **Fibonacci sequence** is calculated from the **sum of the two previous members**. The first two elements are 1, 1. Therefore, the sequence goes like 1, 1, 2, 3, 5, 8, 13, 21, 34…

The following sequence can be generated with an array, but that's easy, so **your task is to implement it recursively**.

If the function **getFibonacci(n)** returns the n^{th} Fibonacci number, we can express it using **getFibonacci(n) = getFibonacci(n-1) + getFibonacci(n-2)**.

However, this will never end, and a Stack Overflow Exception is thrown in a few seconds. For the recursion to stop, it has to have a "bottom". The bottom of the recursion is getFibonacci(1), and should return 1. The same goes for getFibonacci(0).

Input

- On the only line in the input, the user should enter the wanted Fibonacci number **N** where **1 ≤ N ≤ 49.**

Output

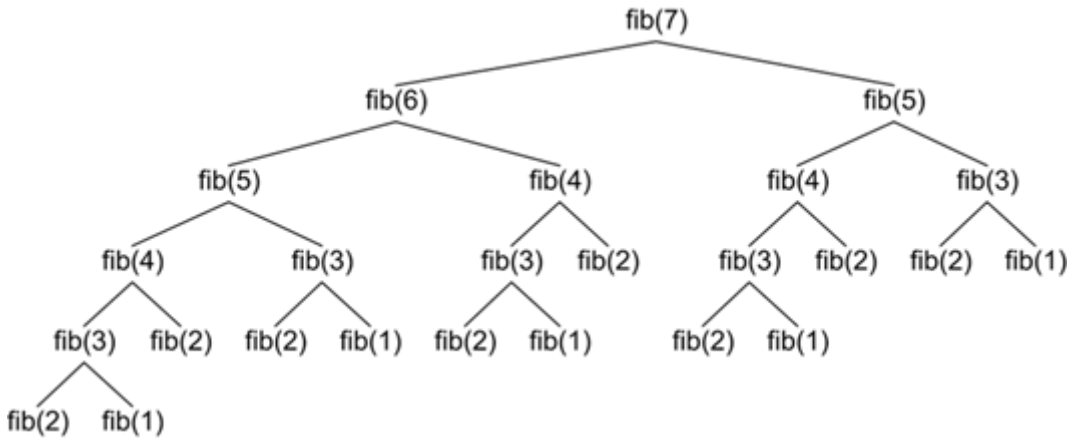
- The output should be the n^{th} Fibonacci number counting from 0.

Examples

Input	Output
5	8
10	89
21	17711

Hint

For the n^{th} Fibonacci number, we calculate the N-1^{st} and the N-2^{nd} number, but for the calculation of N-1^{st} number, we calculate the N-1-1^{st}(N-2^{nd}) and the N-1-2^{nd} number, so we have a lot of repeated calculations.



If you want to figure out how to skip those unnecessary calculations, you can search for a technique called memoization.

7. *Simple Text Editor

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

- "**`1 {string}`**" - **appends** [string] 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, where **1 ≤ N ≤ 105.**
- 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 "**command argument**".
- **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.**

Output

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

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**. Append "abc" Print the third character. Erase 3 characters. Append "xy". Print the second character. Undo the last command - text is now "". Undo the last command - text is now "abc". Print first character.
6 1 Soft 1 Uni 2 1 3 1 1 be 3 2	S o	

8. *Infix to Postfix

Mathematical expressions are **written in an infix notations**, for example "**5 / (3 + 2)**". However, this kind of notation is **not efficient for computer processing**, as you first need to evaluate the expression inside the brackets, so there is a lot of back and forth movement. A more suitable approach is to **convert it into the so-called postfix notations** (also called Reverse Polish Notation), in which the **expression is evaluated from left to right**, for example, "3 2 + 5 /".

Implement an **algorithm that converts** the mathematical expression **from infix notation into a postfix notation**. Use the famous Shunting-yard algorithm.

Input

- You will **receive an expression on a single line consisting of tokens.**
- Tokens could be numbers 0-9, variables a-z, operators +, -, *, / and brackets (or).
- Each token is **separated by exactly one space.**

Output

- The **output should be on a single line**, consisting of **tokens separated by exactly one space**.

Examples

Input	Output
5 / (3 + 2)	5 3 2 + /
1 + 2 + 3	1 2 + 3 +
7 + 13 / (12 - 4)	7 13 12 4 - / +
(3 + x) - y	3 x + y -

9. **Poisonous Plants

You are given **N** plants in a garden. Each of these plants has been added with some amount of pesticide. You are given the pesticide's initial values and each plant's position. After each day, if any plant has more pesticide than the plant at its left, being weaker (more GMO) than the left one, it dies. Print the number of days **after** which no plant dies, i.e. the time after which there are no plants with more pesticide content than the plant to their left.

Input

- The input consists of an integer **N** representing the number of plants.
- The next **single line** consists of **N** integers, where every integer represents each plant's position and amount of pesticides. **1 ≤ N ≤ 100000.**
- Pesticides amount on a plant is between 0 and 1000000000.

Output

- Output a single value equal to the number of days after which no plants die.

Examples

Input	Output	Comments
7 6 5 8 4 7 10 9	2	Initially, all plants are alive. Plants = {(6, 1), (5, 2), (8, 3), (4, 4), (7, 5), (10, 6), (9, 7)} Plants[k] = (i, j) => jth plant has pesticide amount = i. After the 1st day, 4 plants remain as plants 3, 5, and 6 die. Plants = {(6, 1), (5, 2), (4, 4), (9, 7)} After the 2nd day, 3 plants survive as plant 7 dies. Plants = {(6, 1), (5, 2), (4, 4)} After the 3rd day, 3 plants survive, and no more plants die. Plants = {(6, 1), (5, 2), (4, 4)} After the 2nd day, the plants stop dying.
5 7 2 3 9 2	1	

10. **Robotics

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

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

Each robot **processes a product coming from the assembly line**. A **product comes** 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 no free robot** to take it, it should be **queued at the end of the line again**.

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

Input

- On the first line, you will get 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 the format "**`hh:mm:ss`**".
- Next, until the "**End**" command, you will get a product on each line.

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]