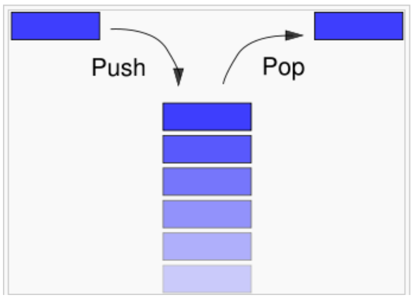
Stack

A **stack** is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

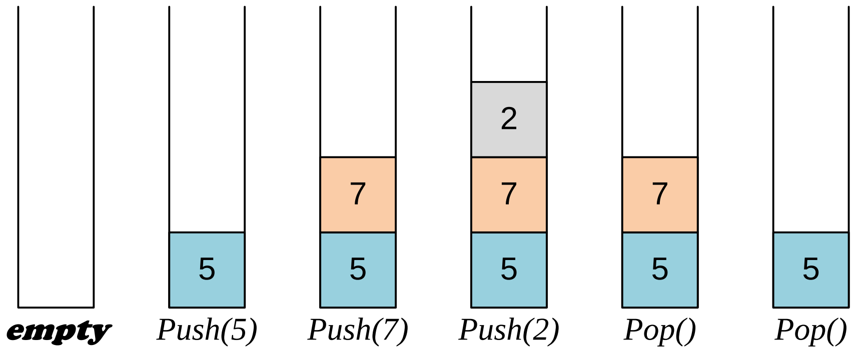
A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the *top* of the stack only.



This feature makes it FILO data structure. FILO stands for ***first-in last-out***. Here, the element which is placed (inserted) first, is accessed last. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

The underlying container may be one of the standard container class template or some other specifically designed container class. This underlying container shall support at least the following operations:

- ***empty*** – test whether container is empty;
- ***size*** – return size;
- ***top*** – access top element
- ***push*** – insert element to the back;
- ***pop*** – remove front element;



Push numbers into the stack and pop them in FILO order:

```
#include <cstdio>
#include <stack>
using namespace std;

int main(void)
{
  // Create an empty stack s.
  stack<int> s;

  // Push to the stack the squares of numbers from 1 to 100.
  for (int i = 1; i <= 10; i++) s.push(i*i);

  // Create stack t that equals to s using copy constructor
  stack<int> t(s);

  // Print the top element and the size of the stack
  printf("Top element is %d\n", t.top());
  printf("Stack size is %d\n", t.size());

  // Print all stack elements, sequentially removing them from the top
  while (!t.empty())
  {
    printf("%d ", t.top());
    t.pop();
  }
  printf("\n");
  return 0;
}
```

E-OLYMP **[6122. Simple stack]** Simulate stack operations.
► Simulate a stack.

E-OLYMP **[5087. Implement a stack]** Simulate push and pop operations.
► Simulate a stack.

E-OLYMP **[5327. Bracket sequence]** The bracket sequence is a correct arithmetic expression, from which all numbers and signs are removed. For example,

$$\mathbf{1 + (((2 + 3) + 5) + (3 + 4)) \rightarrow ((( )) ( ))}$$

Print "YES" if the bracket sequence is correct and "NO" otherwise.

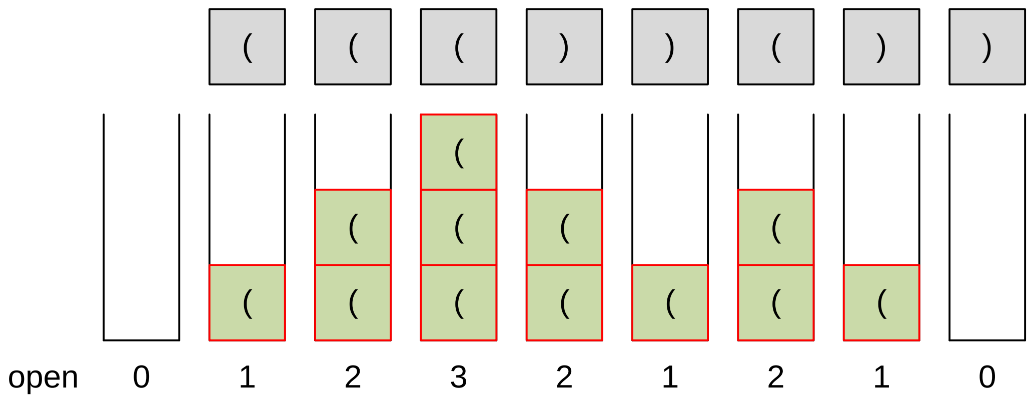
► Declare a stack where we'll save the opening brackets only. When the next symbol arrives, we do the following operation with the stack:

- if the symbol '(' is encountered, then **push** it onto the stack;
- if the symbol ')' is encountered, then **pop** the top element from the stack. If the stack is empty, then the sequence is not parenthetical (at some stage, the number of closing parentheses is greater than the number of opening ones);

At the end of processing the input line, the stack must be empty.

You can simulate the stack using one variable. Let the variable *open* stores the number of open parentheses in the stack. When the '(' symbol is pushed onto the stack, the *open*++ operation is performed. When an element is removed from the top of the stack, operation *open*-- is performed.

Let's simulate the stack for the string "`( ( ( ) ) ( ) )`" from the first sample.



E-OLYMP **2479. Parentheses balance** You are given a string consisting of parentheses ( ) and [ ]. A string of this type is said to be correct:

- if it is the empty string
- if A and B are correct, AB is correct,
- if A is correct, (A) and [A] is correct.

Write a program that takes a sequence of strings of this type and check their correctness. Your program can assume that the maximum string length is 128.

► To solve the problem we'll use stack of characters. We will process sequentially the symbols of the input string and:

- if the current character is an opening parenthesis (round or square), push it into the stack.
- if the current character is a closing bracket, then the corresponding opening parenthesis must be at the top of the stack. If this is not the case, or if the stack is empty, then the expression is not correct.

At the end of processing the correct line, the stack should be empty.

E-OLYMP **5060. Reverse Polish notation** Reverse Polish notation (RPN) is a mathematical notation in which every operator follows all of its operands. It is also known as postfix notation and does not need any parentheses as long as each operator has a fixed number of operands. For example:

- the expression 2 + 4 in RPN is represented like 2 4 +
- the expression 2 * 4 + 8 in RPN is represented like 2 4 * 8 +
- the expression 2 * (4 + 8) in RPN is represented like 2 4 8 + *

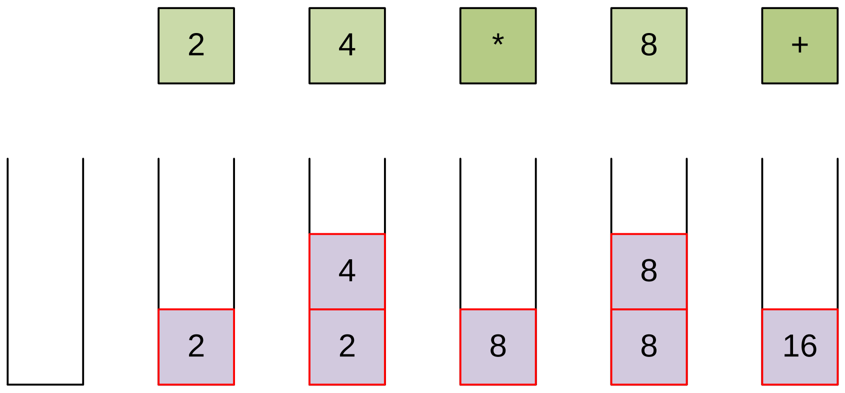
Evaluate the value of an arithmetic expression in Reverse Polish Notation. Valid operators are +, -, *, /. Operator / is an integer division (14 / 3 = 4). Each operand may be an integer or another expression.

► Let's partition the input expression into terms, which are the number or one of the four operators. The terms will be processed as follows:

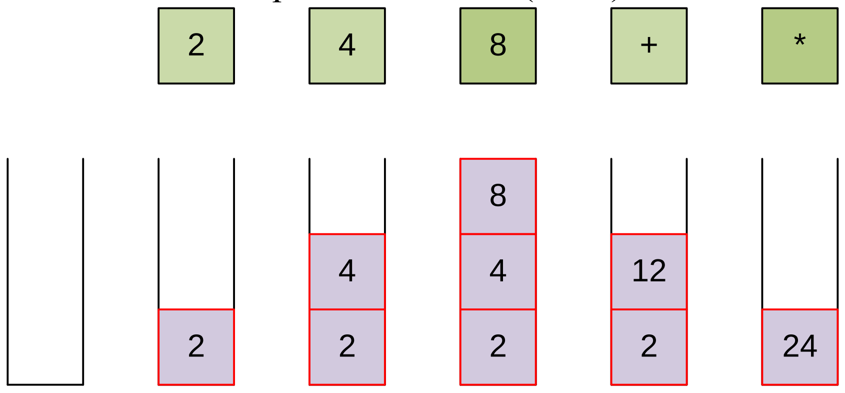
- if term is a *number*, push it into stack;
- if term is an *operator*, extract two numbers from the stack, perform the operation and push the result into stack.

When the expression is processed, the stack contains one number that is the result of calculations.

The expression “2 4 * 8 +” is equivalent to “2 * 4 + 8”.



The expression “2 4 8 + *” is equivalent to “2 * (4 + 8)”.



E-OLYMP **[940. Majority element](#)** Given an array of size n, find the majority element. The majority element is the element that appears more than $\lfloor n/2 \rfloor$ times.

► Let x be the majority element. Let’s start to process the input data. Each number equals to x we shall ***push*** into the stack. If any other number arrives, we shall ***pop*** one number out of stack. At the end of data processing the top of the stack will contain the majority element.

Initially stack is empty. When processing the next element a:

- If stack is empty, then **push**(a);
- If the top of the stack contains number a, then **push**(a);
- If the top of the stack contains number other than a, then **pop**();

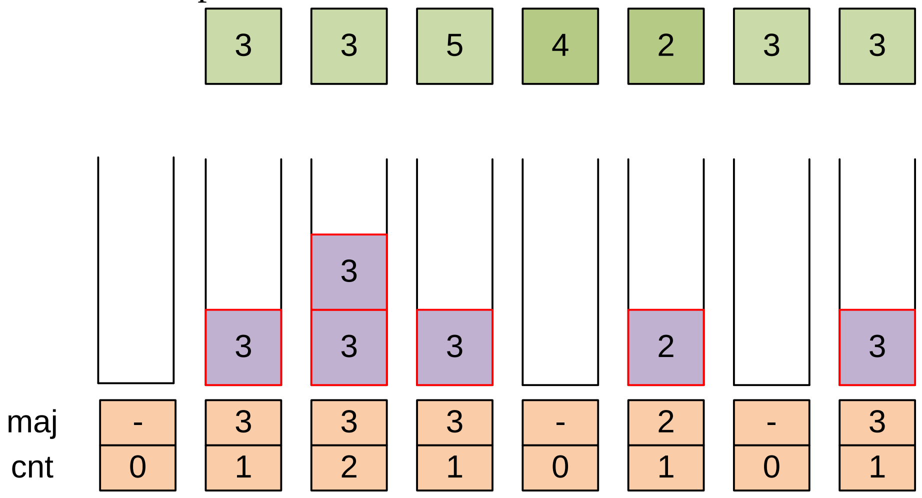
If at the end of processing all the numbers in array, the top of the stack contains some number x (if stack is empty, then there is no majority element), then we must check whether it is a majority element. To do this, it is necessary to calculate how many times the number x occurs in the original array. If x occurs more than $\lfloor n/2 \rfloor$ times, then the answer is positive.

At any time stack contains only one element (possibly multiple times), so let’s simulate the stack with two variables:

- *maj* – number in the stack;

- *cnt* – number of times the number *maj* appears in the stack;

Consider the first sample.



At the end of the algorithm stack contains 3. Let's check whether it is a majority element. To do this, count how many times number 3 occurs in the original array. Number 3 occurs 4 times in the array of length $n = 7$. Since $4 > \lfloor 7/2 \rfloor$, number 3 is a majority element.

Store the input sequence in array m.

```
int m[110];
```

Declare variables *maj* and *cnt* to simulate the stack:

- *maj* – number in the stack;
- *cnt* – number of times the number *maj* appears in the stack;

```
int maj, cnt;
```

Read input numbers into array m.

```
scanf("%d",&n);
for(i = 0; i < n; i++)
  scanf("%d",&m[i]);
```

Initialize stack to be empty.

```
maj = 0; cnt = 0;
```

Process input numbers. Simulate the stack.

```
for(int i = 0; i < n; i++)
{
```

If stack is empty ($cnt = 0$), push m[*i*] into it.

```
  if (cnt == 0) {maj = m[i]; cnt++;}
```

If current element m[*i*] coincides with the element *maj* on the top of the stack, push m[*i*] into stack.

```
  else if (m[i] == maj) cnt++;
```

Otherwise pop element from the stack.

```
  else cnt--;
}
```

Count in variable *cnt* how many times the number *maj* appears in array m.

```
cnt = 0;
for(int i = 0; i < n; i++)
  if (m[i] == maj) cnt++;
```

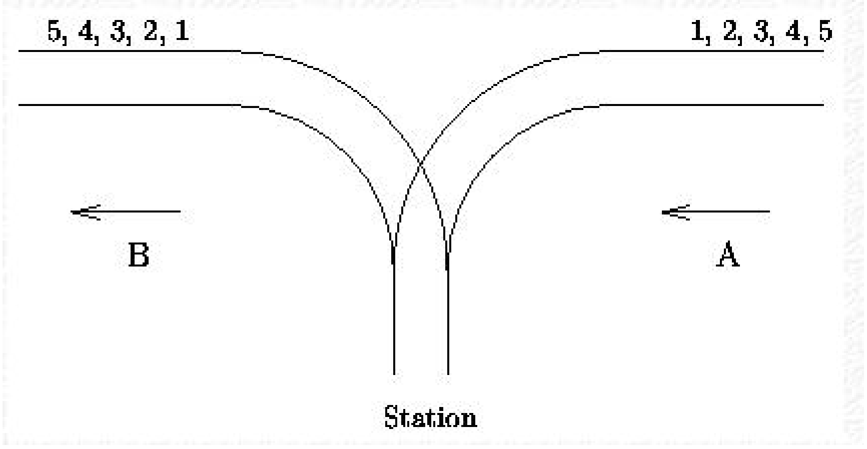
If *cnt* > $\lfloor n/2 \rfloor$, the majority element exists. Otherwise does not exist (assign -1 to *res*).

```
if(2 * cnt > n) res = maj; else res = -1;
```

Print the answer.

```
printf("%d\n",res);
```

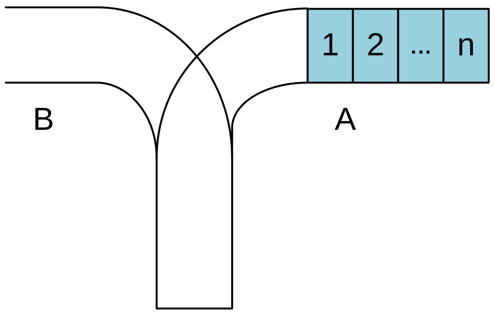
E-OLYMP **1776. Rails** There is a famous railway station in PopPush City. Country there is incredibly hilly. The station was built in the last century. Unfortunately, funds were extremely limited that time. It was possible to establish only a surface track. Moreover, it turned out that the station could be only a dead-end one (see picture) and due to lack of available space it could have only one track.

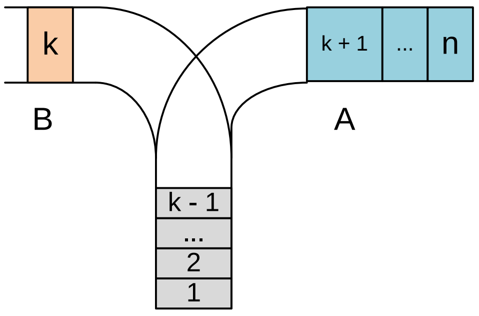


The local tradition is that every train arriving from the direction A continues in the direction B with coaches reorganized in some way. Assume that the train arriving from the direction A has $n \leq 1000$ coaches numbered in increasing order 1, 2, ..., *n*. The chief for train reorganizations must know whether it is possible to marshal coaches continuing in the direction B so that their order will be a_1, a_2, ..., a_n. Help him and write a program that decides whether it is possible to get the required order of coaches. You can assume that single coaches can be disconnected from the train before they enter the station and that they can move themselves until they are on the track in the direction B.

You can also suppose that at any time there can be located as many coaches as necessary in the station. But once a coach has entered the station it cannot return to the track in the direction A and also once it has left the station in the direction B it cannot return back to the station.

► Store the cars that will enter the dead-end station in the stack *s*. On side A, the cars are in the sequence 1, 2, ..., *n*. If the first car on side B should be car *k*, then this can be achieved by driving into a dead end all carriages with numbers 1, 2, ..., *k*, and then moving the car with number *k* to side B.





Let after that the car with number *l* should be the second on side B. If *l* < *k*, then it can be moved in the direction B only if it is currently at the top of the stack *s* (otherwise, the required rearrangement of cars is not possible). If *l* > *k*, then we move from side A all the cars up to the *l*-th iton the stack, and then we move the car *l* to the side B. Continue to simulate the moving of cars in this way until all cars are transported from side A to side B.