PSH: Static Boolean Expression Trees

You will be given symbols printed in a postorder traversal of a boolean expression tree. The internal nodes of the tree will contain one of the following operators: `&` (and), `|` (or), `^` (exclusive-or), or `!` (not). The nodes containing the first three operators will have two children, while the nodes containing the `!` operator will contain only a left child. The leaves of the tree will contain an operand - either `f` for false or `t` for true.

You task is to evaluate the tree, and output either `true` or `false`.

Input Format

The input begins with an integer *t*, which gives the number of test cases in the input.

Each test case will consist of two lines. The first line will contain an integer *n*, which gives the number of nodes in the tree.

The next line will contain *n*, space-separated symbols representing the values from the pots-order traversal of the tree. These will be chosen from the following set of values: {`&`, `|`, `^`, `!`, `t`, `f`}.

Constraints

$1 \leq t \leq 10$

$1 \leq n \leq 10^6$

Output Format

For each test case, output on a line by itself, either `true` or `false`, depending on the value of the tree.

Sample Input 0

```
2
6
t f | f ! ^
8
f t t t ^ | & !
```

Sample Output 0

```
false
true
```

Explanation 0

The trees should be evaluated recursively as follows:

1) A tree of a single node will consist either of 't' or 'f' and should evaluate to true or false, respectively

2) Otherwise the root will contain a boolean operator, not (!), or (|), and(&), or exclusive-or (^).

a) For !, the tree should return the opposite value of its left child.

b) For &, the tree should return the & operator applied to its two children; for |, the | operator applied to the two children; and for ^, the ^ operator applied to its two children.

The first tree in this test case would deserialize to:

```
      ^
    /   \
   |      !
  / \    /
 t   f  f
```

We can step through the evaluation of the tree as follows. First we evaluate the subtree at the bottom left:

```
       ^
     /   \
    t     !
         /
        f
```

Next we evaluate the subtree at the bottom right:

```
       ^
     /   \
    t     t
```

Finally, we can evaluate the remainder of the tree, which would evaluate to:

```
       f
```