All Contests > HourRank 30 > Jerry's Expression

Jerry's Expression

locked

by nikasvanidze

Problem Submissions Leaderboard Discussions Editorial

This problem revolves around the Polish notation.

- *Polish notation is the way to write parenthesis-free expressions. Its distinguishing feature is that it places operators to the left of their operands.*
- *expression* ::= *number* | (*operator expression expression*)
- *operator* ::= $+ \mid - \mid \times \mid \div \mid \dots$
- *For example: "*$(A+B)\times(C-D)$*" is "*$\times + AB - CD$*".*

You are given a Polish notation expression. Operators can be only $+$ and $-$. Each number in expression is replaced with $?$. You have to replace each $?$ with positive integer number, so that value of expression was 0. Also, you have to make the biggest number in expression as small as possible.

Input Format

The only line contains string with expression (string will contain only '?', '+' and '-').

Constraints

- $3 \leq$ *string length* $\leq 10^6$.

Output Format

Return an integer array, k^{th} number should be the number for k^{th} '?' in the string. If there are many solutions print any.

Sample Input 0

```
-?-??
```

Sample Output 0

```
1
2
1
```

Explanation 0

```
- 1 - 2 1   is   1-(2-1) = 0
```

Submissions: 215
Max Score: 45
Difficulty: Medium

Rate This Challenge:
★★★★☆

More

Current Buffer (saved locally, editable)

Rust

```
use std::io::{self, Read};
// use std::collections::HashMap;

#[derive(Clone, Copy, Debug)]
enum Node {
    Op {
        op_type: u8,
        l: usize,
        r: usize,
        accm: i32,
    },
    Val(i32),
    None,
}

impl Default for Node {
    fn default() -> Self { Node::None }
}

fn main() {
    let mut buffer = String::new();
    let stdin = io::stdin();
    let mut handle = stdin.lock();

    handle.read_to_string(&mut buffer).unwrap();

    let mut b = vec![ Node::default(); 10_000_000 ];

    let mut idx_new = 0;

    let mut v = vec![idx_new]; //stack of ops and inputs
    idx_new += 1;

    let mut sign = vec![ 0i8; 10_000_000 ]; //records sign of variables

    let mut var_idxs = vec![];

    for i in buffer.chars() {
        match i {
            '-' | '+' | '?' =>{},
            _ => { break; },
        }
        // println!("stack: {:?}", v );
        let idx = *v.last().unwrap();
        v.pop();
        match i {
            '-' => {
                let n = Node::Op { op_type: '-' as u8, l: idx_new, r: idx_new+1, accm: 0 };
                b[idx] = n;
                v.push(idx_new+1);
                v.push(idx_new);
                idx_new += 2;
            },
            '+' => {
                let n = Node::Op { op_type: '+' as u8, l: idx_new, r: idx_new+1, accm: 0 };
                b[idx] = n;
                v.push(idx_new+1);
                v.push(idx_new);
                idx_new += 2;
            },
            '?' => {
                match b[idx] {
                    Node::None => {
                        b[idx] = Node::Val(1);
                        var_idxs.push(idx);
                    },
                    _ => { panic!("unexpected node type: {:?}", b[idx]); },
                }
            },
            _ => {},
        }
    }
    // println!("");
    // dfs_print( b.as_mut_slice(), 0 );
    // println!("");

    // println!("evaluate..");
    dfs_sum( b.as_mut_slice(), 0 );

    let accumulated = if let Node::Op{ accm, .. } = b[0] {
        // println!("accumulated: {}", accm );
        accm
    } else {
        0
    };

    // dfs_print( b.as_mut_slice(), 0 );
    // println!("");

    // dfs_print_accm( b.as_mut_slice(), 0 );
    // println!("");

    //distribute signs over all variables
    dfs_sign_propagate( b.as_mut_slice(), 0, 1i8, sign.as_mut_slice() );

    let var_neg = var_idxs.iter().filter(|x| sign[**x] < 0i8 ).collect::<Vec<_>>();
    let var_pos = var_idxs.iter().filter(|x| sign[**x] > 0i8 ).collect::<Vec<_>>();
    // println!("neg: {:?}", var_neg);
    // println!("pos: {:?}", var_pos);

    let pos = if accumulated < 0 {
        &var_pos
    } else {
        &var_neg
    };

    let distribute = accumulated.abs() / pos.len() as i32;
    let mut rem = accumulated.abs() % pos.len() as i32;

    for &i in pos.iter() {
        if let &mut Node::Val(ref mut x) = & mut b[*i] {
            *x += distribute;
            if rem > 0 {
                *x += 1;
                rem -= 1;
            }
        }
    }

    for i in var_idxs.iter().filter_map(
        |x| if let Node::Val(y) = b[*x] { Some(y) } else { None }
    ) {
        print!("{} ", i );
    }
    println!("");
}

fn dfs_sign_propagate( n: & mut [Node], idx: usize, sign: i8, sign_array: & mut [i8] ) {
    match n[idx] {
        Node::None => {},
        Node::Op{ op_type, l, r, .. } => {

            let sign_propagate = if op_type as char == '+' {
                sign
            } else {
                -1 * sign
            };

            dfs_sign_propagate( n, l, sign_propagate, sign_array );
            dfs_sign_propagate( n, r, sign_propagate, sign_array );
        },
        Node::Val(_) => {
            sign_array[idx] = sign;
        },
    }
}

fn dfs_sum( n: & mut [Node], idx: usize ) {
    match n[idx] {
        Node::None => {},
        Node::Op{ op_type, l, r, .. } => {

            dfs_sum( n, l );
            dfs_sum( n, r );

            let ( mut sum_l, mut sum_r )  = ( 0i32, 0i32 );

            if let Node::Val(x) = n[l] {
                sum_l += x as i32;
            } else if let Node::Op{accm, ..} = n[l] {
                sum_l += accm;
            }
            if let Node::Val(x) = n[r] {
                sum_r += x as i32;
            } else if let Node::Op{accm, ..} = n[r] {
                sum_r += accm;
            }

            let mut sum;

            if op_type as char == '+' {
                sum = sum_l + sum_r;
            } else {
                sum = sum_l - sum_r;
            }

            // n[idx] = Node::Val(sum);
            n[idx] = Node::Op { op_type: op_type, l: l, r: r, accm: sum };
        },
        Node::Val(_) => {},
    }
}

fn dfs_print( n: & mut [Node], idx: usize ) {
    match n[idx] {
        Node::None => {},
        Node::Op{ op_type, l, r, .. } => {
            print!("(");
            dfs_print( n, l );
            print!("{}", op_type as char );
            dfs_print( n, r );
            print!(")");
        },
        Node::Val(x) => { print!( "{}", x ); },
    }
}

fn dfs_print_accm( n: & mut [Node], idx: usize ) {
    match n[idx] {
        Node::None => {},
        Node::Op{ op_type, l, r, accm } => {
            print!("( {} = ", accm);
            dfs_print_accm( n, l );
            print!("{}", op_type as char );
            dfs_print_accm( n, r );
            print!(")");
        },
        Node::Val(x) => { print!( "{}", x ); },
    }
}
```

Line: 1 Col: 1

Upload Code as File □ Test against custom input

Run Code Submit Code

Contest Calendar | Interview Prep | Blog | Scoring | Environment | FAQ | About Us | Support | Careers | Terms Of Service | Privacy Policy | Request a Feature