

Доктор Пилюлькин изготавливает в лаборатории разные лекарственные вещества. Доктор Пилюлькин почти знает какие компоненты нужны ему для приготовления лекарств, и какие вещества будут получаться в результате реакции. Свои знания Доктор записывает в виде формул химической реакции, которые могут быть неточны. Доктор попросил Незнайку исправить ему формулы для приготовления веществ.

Молекулы состоят из атомов. Атом — это либо одна заглавная латинская буква, либо заглавная латинская буква, за которой идут одна или более строчных латинских букв. Примеры обозначения атомов: H, El, Elerium. Если после обозначения атома записано положительное целое число, оно обозначает число атомов этого типа. Молекула состоит из атомов с указанием числа повторений. Примеры молекул: H_2O , $\text{C}_2\text{H}_5\text{O}_2\text{Li}$. для группировки в записи молекулы могут использоваться круглые или квадратные скобки, но тогда число повторений относится ко всей группе атомов в скобках. Например, $[(\text{CH}_2)_{10}\text{HSi}]_2$.

Формула реакции описывает превращение химических веществ. В процессе реакции одни молекулы превращаются в другие молекулы. Формула реакции записывается в виде $\text{I}_1 + \text{I}_2 + \text{I}_3 \dots = \text{O}_1 + \text{O}_2 \dots$. Молекулы с левой стороны от знака равенства — исходные вещества, а молекулы с правой части — результат реакции. Молекулы в левой и правой части разделяются знаком сложения. И в левой, и в правой части формулы реакции могут находиться одна или более обозначений молекул. Перед обозначением молекулы может быть записано положительное целое число, обозначающее количество молекул в реакции. Пример реакции: $2\text{H}_2 + \text{O}_2 = 2\text{H}_2\text{O}$.

В правильно записанной формуле должен выполняться закон сохранения вещества. Количество атомов каждого вещества в левой части формулы должно быть равно количеству атомов в правой части формулы реакции. Например, реакция $\text{H}_2 + \text{O}_2 = \text{H}_2\text{O}$ или реакция $3\text{H}_2 + \text{CO}_2 = \text{H}_2\text{O}$ записаны неправильно.

Напишите программу, которая из возможно неправильной записи химической реакции получит правильную запись. Преобразованная запись химической реакции должна удовлетворять следующим требованиям:

- Молекулы веществ в левых и правых частях формулы должны следовать в порядке их первого вхождения в левую и правую части соответственно.
- Если одна и та же молекула входит в левую или правую части формулы несколько раз, все вхождения, кроме первого игнорируются. Запись молекулы для этого должна быть посимвольно совпадающей.
- Если в левой или правой части формулы полностью отсутствуют атомы, нужные в другой части, недостающие атомы должны быть записаны в лексикографическом порядке в соответствующей части формулы после веществ из исходной формулы.
- Должен выполняться закон сохранения вещества.
- Числовые коэффициенты перед молекулами должны быть минимально возможными. Поэтому формула реакции $4\text{H}_2 + 2\text{O}_2 = 4\text{H}_2\text{O}$ неправильная.

Формулы реакции вводятся на стандартном потоке ввода по одной формуле на строке. Исправленные формулы должны быть выведены на стандартный поток вывода по одной формуле на строке.

Для представления количества атомов в левой и правой частях формулы и для всех промежуточных вычислений достаточно положительных знаковых 32-битных целых чисел.

Формат ввода:

Формулы реакции вводятся на стандартном потоке ввода по одной формуле на строке. Количество атомов в левой и правой частях и входной формулы, и выходной формулы представимо положительным знаковым 32-битным целым числом. Длина одной входной строки не превышает 100000 байт. Суммарный размер всех входных строк не превышает 1000000 байт.

Формат вывода:

Исправленные формулы должны быть выведены на стандартный поток вывода по одной формуле на строке.

Пример ввода:

```
H2+O2=H2O
Si+O2=SiO2
C+O2+C+O2+C=CO2
(He [C2 (O2Mg) 4 (O2Fe) 2] 2N3) 2H3=(He (C2 (O2Mg) 4 (O2Fe) 2) 2N3) 2H3F4
```

Пример вывода:

```
2H2+O2=2H2O
Si+O2=SiO2
C+O2=CO2
(He [C2 (O2Mg) 4 (O2Fe) 2] 2N3) 2H3+4F=(He (C2 (O2Mg) 4 (O2Fe) 2) 2N3) 2H3F4
```

Код возможного решения

```
use std::{collections::{HashMap, HashSet}, error::Error, fmt::{Display, Write}};

#[derive(Debug, Clone)]
enum Token<'a> {
    End,
    Open(u8),
    Close(u8),
    Equals,
    Atom(&'a str),
    Num(i32),
    Plus,
    Bracket(&'a str, Box<Token<'a>>, &'a str),
    Counted(Box<Token<'a>>, i32),
    Molecule(Vec<Box<Token<'a>>>, &'a str),
    Component(i32, Box<Token<'a>>),
    Side(Vec<Box<Token<'a>>>),
    Equation(Box<Token<'a>>, Box<Token<'a>>),
}

impl<'a> Display for Token<'a> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            Token::Open(c) => f.write_char(*c as char),
            Token::Close(c) => f.write_char(*c as char),
            Token::Equals => f.write_char('='),
            Token::Atom(s) => f.write_str(*s),
            Token::Num(x) => f.write_fmt(format_args!("{}", *x)),
            Token::Plus => f.write_char('+'),
            Token::Bracket(ob, t, cb) => {
                f.write_str(*ob)?;
                t.fmt(f)?;
                f.write_str(*cb)
            },
            Token::Counted(t, cnt) => {
                t.fmt(f)?;
                if *cnt > 1 {
                    f.write_fmt(format_args!("{}", *cnt))?;
                }
            }
        }
    }
}
```

```

        }
        Ok(())
    },
    Token::Molecule(_, s) => {
        f.write_str(*s)?;
        /*
        for i in 0..v.len() {
            v[i].fmt(f)?;
        }
        */
        Ok(())
    },
    Token::Component(cnt, t) => {
        if *cnt > 1 {
            f.write_fmt(format_args!("{}", *cnt))?;
        }
        t.fmt(f)?;
        Ok(())
    },
    Token::Side(v) => {
        for i in 0..v.len() {
            if i > 0 {
                f.write_char('+')?;
            }
            v[i].fmt(f)?;
        }
        Ok(())
    },
    Token::Equation(lhs, rhs) => {
        lhs.fmt(f)?;
        f.write_char('=')?;
        rhs.fmt(f)
    },
    _ => panic!(),
}

}

}

#[derive(Debug, Clone)]
struct ParseError {
    msg: &'static str,
    pos: usize,
}

impl ParseError {
    fn new(msg: &'static str, pos: usize) -> Self {
        Self { msg, pos }
    }
}

impl Display for ParseError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_,>) -> std::fmt::Result {
        f.write_fmt(format_args!("parse error {} at pos {}", self.msg, self.pos))
    }
}

impl Error for ParseError {}

```

```

struct Parser<'a> {
    s: &'a [u8],
    len: usize,
    i: usize,
    t: Token<'a>,
    token_i: usize,
    atom_map: HashMap<&'a str, i32>,
    atom_vec: Vec<&'a str>,
    counts: Vec<Vec<i32>>,
}

impl<'a> Parser<'a> {
    fn new(s: &'a str) -> Self {
        Self {
            s: s.as_bytes(),
            len: s.len(),
            i: 0,
            t: Token::End,
            token_i: 0,
            atom_map: HashMap::new(),
            atom_vec: Vec::new(),
            counts: vec![Vec::new(), Vec::new()],
        }
    }

    fn next_token(&mut self) -> Result<(), Box<dyn Error>> {
        self.token_i = self.i;
        if self.i >= self.len {
            self.t = Token::End;
        } else if self.s[self.i] >= b'0' && self.s[self.i] <= b'9' {
            let mut val = 0;
            while self.i < self.len && self.s[self.i] >= b'0' && self.s[self.i] <= b'9' {
                val = val * 10 + (self.s[self.i] - b'0') as i32;
                self.i += 1;
            }
            self.t = Token::Num(val);
        } else if self.s[self.i] >= b'A' && self.s[self.i] <= b'Z' {
            let beg = self.i;
            self.i += 1;
            while self.i < self.len && self.s[self.i] >= b'a' && self.s[self.i] <= b'z' {
                self.i += 1;
            }
            self.t = Token::Atom(std::str::from_utf8(&self.s[beg..self.i]).unwrap());
        } else if self.s[self.i] == b'(' || self.s[self.i] == b'[' {
            self.t = Token::Open(self.s[self.i]);
            self.i += 1;
        } else if self.s[self.i] == b')' || self.s[self.i] == b']' {
            self.t = Token::Close(self.s[self.i]);
            self.i += 1;
        } else if self.s[self.i] == b'=' {
            self.t = Token::Equals;
            self.i += 1;
        } else if self.s[self.i] == b'+' {
            self.t = Token::Plus;
            self.i += 1;
        } else {
            return Err(Box::new(ParseError::new("invalid char", self.i)));
        }
    }
}

```

```

    }
    Ok(())
}

fn counted(&mut self) -> Result<Token<'a>, Box<dyn Error>> {
    let b: Box<Token<'a>>;
    match self.t {
        Token::Atom(_) => {
            b = Box::new(self.t.clone());
            self.next_token()?;
        },
        Token::Open(oc) => {
            let ob = std::str::from_utf8(&self.s[self.i-1..self.i]).unwrap();
            self.next_token()?;
            let m = self.molecule()?;
            if let Token::Close(cc) = self.t {
                if (oc == b'(' && cc == b')') || (oc == b'[' && cc == b']') {
                    let cb = std::str::from_utf8(&self.s[self.i-1..self.i]).unwrap();
                    self.next_token()?;
                    b = Box::new(Token::Bracket(ob, Box::new(m), cb));
                } else {
                    return Err(Box::new(ParseError::new("bracket mismatch", self.i)));
                }
            } else {
                return Err(Box::new(ParseError::new("expect closing bracket", self.i)));
            }
        },
        _ => {
            return Err(Box::new(ParseError::new("unexpected token", self.i)));
        },
    }
    let mut cnt = 1;
    if let Token::Num(cc) = self.t {
        cnt = cc;
        self.next_token()?;
    }
    Ok(Token::Counted(b, cnt))
}

fn molecule(&mut self) -> Result<Token<'a>, Box<dyn Error>> {
    let mut v: Vec<Box<Token<'a>>> = Vec::new();
    let beg = self.token_i;
    loop {
        match self.t {
            Token::Open(_) | Token::Atom(_) => v.push(Box::new(self.counted()?)),
            _ => break,
        }
    }
    let end = self.token_i;
    Ok(Token::Molecule(v, std::str::from_utf8(&self.s[beg..end]).unwrap()))
}

fn component(&mut self) -> Result<Token<'a>, Box<dyn Error>> {
    let mut koef = 1;
    if let Token::Num(kk) = self.t {
        koef = kk;
        self.next_token()?;
    }
}

```

```

        let m = self.molecule()?;
        Ok(Token::Component(koef, Box::new(m)))
    }
fn side(&mut self) -> Result<Token<'a>, Box<dyn Error>> {
    let mut comps: Vec<Box<Token<'a>>> = Vec::new();
    comps.push(Box::new(self.component()?));
    while let Token::Plus = self.t {
        self.next_token()?;
        comps.push(Box::new(self.component()?));
    }
    Ok(Token::Side(comps))
}
fn equation(&mut self) -> Result<Token<'a>, Box<dyn Error>> {
    let s1 = self.side()?;
    if let Token::Equals = self.t {
        self.next_token()?;
        let s2 = self.side()?;
        Ok(Token::Equation(Box::new(s1), Box::new(s2)))
    } else {
        Err(Box::new(ParseError::new("= expected", self.i)))
    }
}
fn parse(&mut self) -> Result<Token<'a>, Box<dyn Error>> {
    self.next_token()?;
    let root = self.equation()?;
    if let Token::End = self.t {
        Ok(root)
    } else {
        Err(Box::new(ParseError::new("EOF expected", self.i)))
    }
}

fn reset_component_count(&mut self, node: &mut Token<'a>) {
    match node {
        Token::Component(cnt, _) => {
            *cnt = 1;
        },
        Token::Side(v) => {
            for item in v {
                self.reset_component_count(item.as_mut());
            }
        },
        Token::Equation(lhs, rhs) => {
            self.reset_component_count(lhs.as_mut());
            self.reset_component_count(rhs.as_mut());
        },
        _ => panic!(),
    }
}

fn set_component_count(&mut self, node: &mut Token<'a>, cnts: &[i32]) -> usize {
    match node {
        Token::Component(cnt, _) => {
            *cnt = cnts[0];
            return 1;
        },
    }
}

```

```

Token::Side(v) => {
    let mut res = 0;
    for t in v {
        res += self.set_component_count(t.as_mut(), &cnts[res..]);
    }
    return res;
},
Token::Equation(lhs, rhs) => {
    let cnt = self.set_component_count(lhs.as_mut(), cnts);
    let cnt2 = self.set_component_count(rhs.as_mut(), &cnts[cnt..]);
    return cnt + cnt2;
},
_ => panic!(),
}

fn remove_duplicates(&mut self, node: &mut Token<'a>) {
    match node {
        Token::Side(v) => {
            let mut ms: HashSet<&'a str> = HashSet::new();
            let mut j = 0;
            for i in 0..v.len() {
                let m: &'a str;
                if let Token::Component(_, mol) = v[i].as_mut() {
                    if let Token::Molecule(_, mm) = mol.as_mut() {
                        m = *mm;
                    } else {
                        panic!();
                    }
                } else {
                    panic!();
                }
                if !ms.contains(m) {
                    ms.insert(m);
                    v.swap(i, j);
                    j += 1;
                }
            }
            v.resize_with(j, || Box::new(Token::End));
        },
        Token::Equation(lhs, rhs) => {
            self.remove_duplicates(lhs.as_mut());
            self.remove_duplicates(rhs.as_mut());
        },
        _ => panic!(),
    }
}

fn collect_atoms(&mut self, node: &Token<'a>, am: &mut HashMap<&'a str, i32>,
av: &mut Vec<&'a str>) {
    match node {
        Token::Atom(s) => {
            am.entry(*s).or_insert_with(|| {
                let len = av.len() as i32;
                av.push(*s);
                len
            });
        },
    }
}

```

```

    },
    Token::Bracket(_, t, _) => {
        self.collect_atoms(t.as_ref(), am, av);
    },
    Token::Counted(t, _) => {
        self.collect_atoms(t, am, av);
    },
    Token::Molecule(v, _) => {
        for i in 0..v.len() {
            self.collect_atoms(&v[i], am, av);
        }
    },
    Token::Component(_, t) => {
        self.collect_atoms(t.as_ref(), am, av);
    },
    Token::Side(v) => {
        for i in 0..v.len() {
            self.collect_atoms(&v[i], am, av);
        }
    },
    Token::Equation(lhs, rhs) => {
        self.collect_atoms(lhs.as_ref(), am, av);
        self.collect_atoms(rhs.as_ref(), am, av);
    },
    _ => (),
}

}

fn compute_presence(&mut self, node: &Token<'a>, flag: &mut[bool]) {
    match node {
        Token::Atom(a) => {
            flag[self.atom_map[*a] as usize] = true;
        },
        Token::Bracket(_, t, _) => {
            self.compute_presence(t.as_ref(), flag);
        },
        Token::Counted(t, _) => {
            self.compute_presence(t.as_ref(), flag);
        },
        Token::Molecule(v, _) => {
            for i in 0..v.len() {
                self.compute_presence(v[i].as_ref(), flag);
            }
        },
        Token::Component(_, t) => {
            self.compute_presence(t.as_ref(), flag);
        },
        Token::Side(v) => {
            for i in 0..v.len() {
                self.compute_presence(v[i].as_ref(), flag);
            }
        },
        _ => panic!(),
    }
}

}

fn do_insert_missing(&mut self, node: &mut Token<'a>, p: &[bool]) {
    match node {

```



```

Token::Side(v) => {
    for i in 0..p.len() {
        if !p[i] {
            let a = Box::new(Token::Atom(self.atom_vec[i]));
            let c = Box::new(Token::Counted(a, 1));
            let m = Box::new(Token::Molecule(vec![c], self.atom_vec[i]));
            let cc = Box::new(Token::Component(1, m));
            v.push(cc);
        }
    }
},
_ => panic!(),
}

fn insert_missing(&mut self, node: &mut Token<'a>) {
    match node {
        Token::Equation(lhs, rhs) => {
            let mut lhp = vec![false; self.atom_vec.len()];
            self.compute_presence(lhs.as_ref(), &mut lhp);
            let mut rhp = vec![false; self.atom_vec.len()];
            self.compute_presence(rhs.as_ref(), &mut rhp);
            self.do_insert_missing(lhs.as_mut(), &lhp);
            self.do_insert_missing(rhs.as_mut(), &rhp);
        },
        _ => panic!(),
    }
}

fn count_atoms(&mut self, node: &Token<'a>, cnt: &mut [i32], mult: i32) {
    match node {
        Token::Atom(s) => cnt[self.atom_map[*s] as usize] += mult,
        Token::Bracket(_, t, _) => self.count_atoms(t.as_ref(), cnt, mult),
        Token::Counted(t, v) => self.count_atoms(t.as_ref(), cnt, mult * *v),
        Token::Molecule(v, _) => {
            for i in 0..v.len() {
                self.count_atoms(v[i].as_ref(), cnt, mult);
            }
        },
        Token::Component(v, t) => self.count_atoms(t.as_ref(), cnt, mult * *v),
        _ => panic!(),
    }
}

fn count_atoms_side(&mut self, node: &Token<'a>, cnt: &mut Vec<Vec<i32>>>) {
    match node {
        Token::Side(v) => {
            for i in 0..v.len() {
                let mut cc = vec![0; self.atom_vec.len()];
                self.count_atoms(v[i].as_ref(), &mut cc, 1);
                cnt.push(cc);
            }
        },
        _ => panic!(),
    }
}

fn count_atoms_equation(&mut self, node: &Token<'a>) {
    match node {
        Token::Equation(lhs, rhs) => {

```

```

        let mut lc: Vec<Vec<i32>> = Vec::new();
        self.count_atoms_side(lhs.as_ref(), &mut lc);
        let mut rc: Vec<Vec<i32>> = Vec::new();
        self.count_atoms_side(rhs.as_ref(), &mut rc);
        self.counts = Vec::new();
        for v in lc {
            self.counts.push(v);
        }
        for mut v in rc {
            for x in &mut v {
                *x = -*x;
            }
            self.counts.push(v);
        }
    },
    _ => panic!(),
}
}

fn apply(mat: &Vec<Vec<i32>>, v: &[i32], out: &mut [i32]) {
    // mat: n rows; m columns; v: n; out: m
    out.fill(0);
    let rows = mat.len();
    let cols = out.len();
    for r in 0..rows {
        for c in 0..cols {
            out[c] += mat[r][c] * v[r];
        }
    }
}

fn is_zero(v: &[i32]) -> bool {
    for &x in v.iter() {
        if x != 0 {
            return false;
        }
    }
    true
}

fn apply_bool(mat: &Vec<Vec<i32>>, v: &[i32], out: &mut [i32]) -> bool {
    Self::apply(mat, v, out);
    Self::is_zero(out)
}

fn next_vector(v: &mut [i32], lim: &mut i32, limcnt: &mut usize) -> bool {
    if *limcnt == v.len() {
        v.fill(1);
        *lim += 1;
        *limcnt = 0;
    }
    loop {
        let mut i = 0;
        while i < v.len() {
            if v[i] == *lim {
                *limcnt -= 1;
                v[i] = 1;
            } else {
                v[i] += 1;
                if v[i] == *lim {

```

```

        *limcnt += 1;
    }
    break;
}
i += 1;
}
if *limcnt > 0 {
    break;
}
}
false
}
}

fn solve(s : &str) -> Result<String, Box<dyn std::error::Error>> {
    let mut p = Parser::new(s);
    let mut root = p.parse()?;
    //println!("Parsed: {}", root);
    p.reset_component_count(&mut root);
    //println!("Reset count: {}", root);
    p.remove_duplicates(&mut root);
    //println!("Remove dups: {}", root);
    let mut atom_map: HashMap<&str, i32> = HashMap::new();
    let mut atom_vec: Vec<&str> = Vec::new();
    p.collect_atoms(&root, &mut atom_map, &mut atom_vec);
    atom_vec.sort();
    atom_map.clear();
    for i in 0..atom_vec.len() {
        atom_map.insert(atom_vec[i], i as i32);
    }
    //println!("map: {:?}", atom_map);
    //println!("vec: {:?}", atom_vec);
    p.atom_map = atom_map;
    p.atom_vec = atom_vec;
    p.insert_missing(&mut root);
    //println!("Append missing: {}", root);
    p.count_atoms_equation(&root);
    //println!("Counts: {:?}", p.counts);
    let mut cnts = vec![1; p.counts.len()];
    let mut lim = 1;
    let mut limcnt = p.counts.len();
    let mut out = vec![0; p.counts[0].len()];
    loop {
        if Parser::apply_bool(&p.counts, &cnts, &mut out) {
            p.set_component_count(&mut root, &cnts);
            return Ok(format!("{}", root));
        }
        if Parser::next_vector(&mut cnts, &mut lim, &mut limcnt) {
            return Ok("BAD".to_string());
        }
    }
}

fn main() -> Result<>, Box<dyn std::error::Error>> {
    let mut buf = String::new();
    while std::io::stdin().read_line(&mut buf)? > 0 {

```

```
        println!("{}", solve(buf.trim()));  
        buf.clear();  
    }  
    Ok(())  
}
```