

Незнайка и эпидемия. 11 класс

Цветочный город поразила эпидемия загадочного заболевания пурпурного пятого пальца. Доктору Пилюлькину удалось определить, что заболевание имеет вирусную природу и передается при непосредственном контакте мизинцами. Удалось также выявить «нулевого» пациента, который приехал зараженным из Зеленого города, и отследить все контакты, приведшие к заражениям.

Вирус болезни кодирует свою наследственную информацию в РНК длины оснований. Было установлено, что при каждом заражении происходила мутация ровно одного из оснований РНК, и было установлено положение мутировавшего основания в РНК. РНК вируса можно представить как строку длины, состоящую из 4 букв A, C, G, U .

Если бы мутаций не было, то у всех заболевших РНК вируса полностью совпадали, но каждая мутация приводит к тому, что неизменная часть РНК вируса становится все меньше и меньше. Напишите программу, которая для двух пациентов определит длину самого большого совпадающего фрагмента РНК вирусов, заразивших этих пациентов. Например, если РНК вируса первого пациента кодируется как $AAGCUUCAG$, а РНК вируса второго пациента кодируется как $ACGCUUAAG$, то у этих РНК есть три совпадающих фрагмента: A , $GCUU$ и AG . Самый длинный фрагмент имеет длину 4. Обратите внимание, что фрагменты в двух РНК должны начинаться на тех же самых местах.

У доктора Пилюлькина компьютер, к сожалению, достаточно старый, поэтому ваша программа должна экономно использовать ресурсы. Обратите внимание на ограничения по памяти в этой задаче. Memory RSS limit: 128Mb Stack limit: 8Mb.

Формат ввода:

Первая строка входных данных содержит три целых положительных числа M, N, K . M – это длина РНК вируса ($M < 801$), N – количество пациентов ($N < 1000001$), K – количество запросов на определение длины ($K < 1001$). Следующие $N - 1$ строк содержат описания контактов, приведших к заражению. Каждый контакт – это три числа S, D, B , обозначающие, что пациент S заразил пациента D (пациенты нумеруются подряд от 1 до N включительно), при этом произошла мутация в основании B (основания нумеруются подряд от 0 до $M - 1$ включительно). Гарантируется, что контакты описывают одно дерево заражений, в котором «нулевой» пациент имеет номер 1. Следующие K строк содержат описания запросов в виде пар чисел P, Q , где P и Q – номера пациентов, для которых нужно определить гарантированную длину самого большого совпадающего фрагмента РНК. Поскольку не даны сами РНК вирусов, а только места мутаций, возможно, что совпадающие фрагменты могут быть и длиннее за счет мутаций, которые компенсируют друг друга. Это учитывать не нужно.

Формат вывода:

Для каждого запроса выведите одно число – длину самого большого совпадающего фрагмента РНК.

Пример ввода:

```
16 3 1
1 2 6
2 3 12
3 1
```

Пример вывода:

```
6
```

Код возможного решения

```
use std::error::Error;
```

```

pub struct State {
    pub n: usize, // number of nodes; nodes are numbered 0..n
    pub m: usize, // size of the DNA
    uplink: Vec<u32,u32>,
    downlink: Vec<Vec<u32, u32>>,
    first_entry: Vec<u32>,
    depths: Vec<u32>,
    depth_nodes: Vec<u32>,
    rmq_idx: Vec<Vec<u32>>,
}

impl State {
    fn new(n: usize, m: usize, uplink: Vec<(u32,u32)>, downlink: Vec<Vec<(u32, u32)>>)
    -> Self {
        Self {
            n,
            m,
            uplink,
            downlink,
            first_entry: vec![u32::MAX; n],
            depths: Vec::new(),
            depth_nodes: Vec::new(),
            rmq_idx: Vec::new(),
        }
    }

    fn make_depths(&mut self) {
        let mut stk: Vec<(u32, u32, u32)> = Vec::new(); // (node, index, depth)
        stk.push((0, 0, 0));
        while stk.len() > 0 {
            let l = stk.last_mut().unwrap();
            let node = l.0 as usize;
            let index = l.1 as usize;
            let depth = l.2;
            if self.first_entry[node] == u32::MAX {
                self.first_entry[node] = self.depths.len() as u32;
            }
            self.depths.push(l.2);
            self.depth_nodes.push(l.0);
            if index >= self.downlink[node].len() {
                _ = stk.pop();
            } else {
                l.1 += 1;
                stk.push((self.downlink[node][index].0, 0, depth + 1));
            }
        }
    }

    fn build_rmq(&mut self) {
        let mut width = 1_usize;
        let dlen = self.depths.len();
        self.rmq_idx.push((0..dlen as u32).collect::<Vec<u32>>());
        loop {
            let newwidth = width * 2;
            if newwidth > dlen {
                break;
            }
        }
    }
}

```

```

    }

    let prev = self.rm_idx.last().unwrap();
    let mut cur: Vec<u32> = Vec::new();
    for i in 0..dlen {
        let mut idx = prev[i];
        if i + width < dlen && self.depths[prev[i+width] as usize]
            < self.depths[idx as usize] {
            idx = prev[i+width];
        }
        cur.push(idx);
    }
    self.rm_idx.push(cur);
    width = newwidth;
}

}

fn query_rm_idx(&self, from: usize, to: usize) -> usize {
    let width = to - from + 1;
    if width == 1 {
        self.rm_idx[0][from] as usize
    } else if width == 2 {
        self.rm_idx[1][from] as usize
    } else {
        let mut p2 = width.next_power_of_two();
        if width == p2 {
            let idx = p2.trailing_zeros();
            self.rm_idx[idx as usize][from] as usize
        } else {
            let idx = p2.trailing_zeros() - 1;
            p2 >>= 1;
            let mut minidx = self.rm_idx[idx as usize][from] as usize;
            let otheridx = self.rm_idx[idx as usize][to + 1 - p2] as usize;
            if self.depths[otheridx] < self.depths[minidx] {
                minidx = otheridx;
            }
            minidx
        }
    }
}

}

fn query_lca(&self, node1: u32, node2: u32) -> u32 {
    let left = std::cmp::min(self.first_entry[node1 as usize],
        self.first_entry[node2 as usize]);
    let right = std::cmp::max(self.first_entry[node1 as usize],
        self.first_entry[node2 as usize]);
    let idx = self.query_rm_idx(left as usize, right as usize);
    self.depth_nodes[idx]
}

fn collect_mutations(&self, mut from: u32, to: u32, out: &mut Vec<u32>) {
    while from != to {
        out.push(self.uplink[from as usize].1);
        assert!(self.uplink[from as usize].0 != from);
        from = self.uplink[from as usize].0;
    }
}

```

```

}

fn calc_common(&self, mutts: &mut Vec<u32>) -> u32 {
    if mutts.len() == 0 {
        return self.m as u32;
    }
    mutts.sort_unstable();
    mutts.dedup();
    let mut longest = mutts[0];
    for i in 1..mutts.len() {
        longest = std::cmp::max(longest, mutts[i]-mutts[i-1]-1);
    }
    std::cmp::max(longest, self.m as u32 -*mutts.last().unwrap()-1)
}

fn find_common(&self, n1: u32, n2: u32) -> u32 {
    if n1 == n2 {
        return self.m as u32;
    }
    let lca = self.query_lca(n1, n2);
    let mut mutts = vec![];
    if lca == n1 {
        self.collect_mutations(n2, lca, &mut mutts);
    } else if lca == n2 {
        self.collect_mutations(n1, lca, &mut mutts);
    } else {
        self.collect_mutations(n1, lca, &mut mutts);
        self.collect_mutations(n2, lca, &mut mutts);
    }
    self.calc_common(&mut mutts)
}

}

fn main() -> Result<(), Box<dyn Error>> {
    let mut buf = String::new();
    _ = std::io::stdin().read_line(&mut buf)?;
    let mut iter = buf.split_ascii_whitespace();
    let m = iter.next().ok_or("")?.parse::<usize>()?;
    let n = iter.next().ok_or("")?.parse::<usize>()?;
    let k = iter.next().ok_or("")?.parse::<usize>()?;
    let mut uplink: Vec<(u32,u32)> = vec![(0, 0); n];
    let mut downlink: Vec<Vec<(u32, u32)>> = vec![vec![]; n];
    for _ in 0..n-1 {
        buf.clear();
        _ = std::io::stdin().read_line(&mut buf)?;
        let mut iter = buf.split_ascii_whitespace();
        let s = iter.next().ok_or("")?.parse::<u32>()?;
        let d = iter.next().ok_or("")?.parse::<u32>()?;
        let b = iter.next().ok_or("")?.parse::<u32>()?;
        uplink[d as usize - 1] = (s - 1, b);
        downlink[s as usize - 1].push((d - 1, b));
    }

    let mut st = State::new(n, m, uplink, downlink);
    st.make_depths();
    st.build_rmq();
}

```

```

//println!("depths: {:?}", st.depths);
//println!("depth_nodes: {:?}", st.depth_nodes);
//println!("rmq: {:?}", st.rm_idx);

for _ in 0..k {
    buf.clear();
    _ = std::io::stdin().read_line(&mut buf)?;
    let mut iter = buf.split_ascii_whitespace();
    let p = iter.next().ok_or("")?.parse::<u32>()?;
    let q = iter.next().ok_or("")?.parse::<u32>()?;
    println!("{}", st.find_common(p-1, q-1));
}

Ok(())
}

```