Editorial

Problem A

Very simple problem. It is quite simple to come up with an $O(1)$ formula giving the number of times the i-th finger is counted.

Problem B

It will be possible to break all friendships if and only if we can split the kids into two sets so that no two kids on the same set are friends. This is the same as asking whether the graph representing the friendships is bipartite.

Problem C

Notice that a level actually represents a permutation of the positions. For example:

```
1 2 3 4 5
|-| | |-|
| |-| |-|
|-| |-| |
| | | |-|
3 2 4 5 1
```

So this level corresponds to the permutation $(3, 2, 4, 5, 1)$. What we are asking is what is the minimum number of horizontal sections we need in a level to have it correspond to the same permutation. Each such section swaps two adjacent elements so in other words, we are asking what is the minimum number of adjacent swaps we need to perform in order to obtain the permutation that corresponds to the given level.

This is the number of inversions on that permutation because we can always obtain a given permutation by performing a number of adjacent swaps equal to the number of inversions.

The reason this is true is that given an array with inversions, there is always an adjacent inversion. So we can build a level with a minimum number of horizontal

sections by starting with the target permutation and adding horizontal sections on adjacent inversions iteratively:

```
3 2 4 5 1
|-| | | |  A
2 3 4 5 1
| | | |-|  B
2 3 4 1 5
| | |-| |  C
2 3 1 4 5
| |-| | |  D
2 1 3 4 5
|-| | | |  E
1 2 3 4 5
```

We can get the level by reversing:

```
|-| | | | E
| |-| | | D
| | |-| | C
| | | |-| B
|-| | | | A
```

The number of inversions can be easily counted in $O(n \cdot \log(n))$ using merge sort.

Problem D

This problem was actually supposed to be harder than this. The day before the contests I realized that it was easy to solve using a randomized algorithm.

Computing the real value of both sides of the equality and checking whether it holds will TLE because the number are too big.

However if $y^u \cdot n^o = a^c$ then it must also hold $(y^u \cdot n^o) \mod m = a^c \mod m$ for all $m \geq 1$. Given a fixed m this equality can be checked efficiently using fast exponentiation. Because the numbers don't get bigger than m this will run quickly. So we can simply generate some random m's and check whether $(y^u \cdot n^o) \mod m = a^c \mod m$ holds for all of them. We output yes if so and no otherwise.

I will let you think about why this works with high probability. But the idea is that if $y^u \cdot n^o \neq a^c$ then write $x = y^u \cdot n^o - a^c$. A random m will be a false positive iff m is a divisor of x. If it was easy to randomly generate a divisor of a given integer then factoring integers would be easy.

If we generate a test case where y, u, n, o, a, c are so that $y^u \cdot n^o - a^c = n!$ for some big n then every $m \leq n$ would be a false positive. The maximum value of

$y^u \cdot n^o$ is about $2^{2^{71}}$. And $\log(\log((2^{29})!) \approx 71$ so there is a possibility that such a test case exists. But even this can be overcome by generating large enough values of m.

Intended solution:

One way to solve this would be to factor y, n and a into prime numbers and then check the equality using the exponents. For example, if $y = 10, u = 4, n = 245, o = 2, a = 24500$ and $c = 2$ we can factor y, n and a into:

$$y = 10 = 2 \cdot 5 \quad , \quad n = 245 = 7^2 \cdot 5 \quad \text{and} \quad a = 24500 = 2^2 \cdot 5^3 \cdot 7^2$$

So we have

$$y^u \cdot n^o = 2^u \cdot 5^{u+o} \cdot 7^{2o} \quad \text{and} \quad a^c = 2^{2c} \cdot 5^{3c} \cdot 7^{2c}$$

Therefore equality holds iff the exponents are equal on both sides. That is:

$$u = 2c \quad , \quad u + o = 3c \quad \text{and} \quad 2o = 2c$$

These equalities are easy to check as they don't involve large numbers anymore. In this case it is true $u = 4 = 2c$, $u + o = 4 + 2 = 6 = 3c$ and $2o = 4 = 2c$.

But the input numbers are too large to allow such a factorization. However, what makes checking the exponents work is not the fact that we expresses the bases as prime numbers. The important thing is to express them as products of any set of *coprime* numbers.

For example let $B = \{2, 5, 49\}$. These are not prime numbers as 49 is not prime. However all the numbers in B are pairwise coprime. Thus if we can express y, n and a as products of B then we will be able to check equality in the same way using the exponents.

With this set we would get:

$$y = 10 = 2 \cdot 5 \quad , \quad n = 49^1 \cdot 5 \quad \text{and} \quad a = 24500 = 2^2 \cdot 5^3 \cdot 49^1$$

So

$$y^u \cdot n^o = 2^u \cdot 5^{u+o} \cdot 49^o \quad \text{and} \quad a^c = 2^{2c} \cdot 5^{3c} \cdot 49^c$$

Therefore equality holds iff the exponents are equal on both sides. That is:

$$u = 2c \quad , \quad u + o = 3c \quad \text{and} \quad o = c$$

Such a set is called a *gcd-free basis*. It turns out that computing such a basis can be done in polynomial time.

You can find here a description of how to compute this.

Problem E

Let $t[i]$ be the time at which the i-th cars meets the $i+1$-th car. We set $t[i] = \infty$ if it never meets the next car. The answer to a query at time T is simply the number of indexes such that $t[i] < T$ because one a car meets the next car, they merge into a single group. Thus the number of groups is equal to the number of cars who did not meet the next car. This can be solved efficiently using binary search on the sorted array t.

It remains to compute the array t. There are several ways to do this. One way is a kind of dynamic programming to compute the speed functions of each car.

Let n be the number of cars and s the speeds of the cards. For the last car the function is trivial: constant speed equal to $s[n-1]$. For car $n-2$ the function is equal to $s[n-2]$ from 0 to $t[n-2]$ and then equal to $s[n-1]$, unless $t[n-2] = \infty$ in which case it is constantly equal to $s[n-2]$. In general:

$$speed(i, T) = \begin{cases} s[i] & \text{if } T \leq t[i] \\ speed(i+1, T) & \text{if } n \text{ otherwise} \end{cases}$$

These functions can be build from right to left in $O(n)$.

Now, $t[i]$ will be such that

$$\int_0^{t[i]} speed(i, x)\ dx = d[i+1] - d[i]$$

When we build the functions we will also keep track of the values of the areas in an array to be able to query them efficiently as cumulative sum queries. It is not too hard to implement this to get an $O(n \cdot \log(n))$ runtime.