BUET CSE 17 Reception Contest
Editorial

Boring Marriage

Problemsetter: Rafid Bin Mostafa

This problem is so easy that the author was too lazy to write an editorial…..

Lover Numbers

Problemsetter: Saifullah Talukder

Let's think about two lovers.. Let's look at their binary representation. If a bit of a lover is 1 then at the same bit position of it's lover must be 0. Because if it's 1 then after xor it will become 0. After the summation that bit will also be 0 but with a carry. Hence we can't have 1 at the same bit position of two lovers. So the possible combinations of bit of two lovers are 1,0 ; 0,1 and 0,0.

Now, for the largest lover not exceeding X (let it be Y) we can choose the bit pattern greedily. If a bit of X is 0 then at that position Y will have 1 (we won't take 0 because we want the largest number). And if it's 1 then we will make it 0.Simply put, we just have to flip the bits of X.

For the smallest lover greater than X (let it be Z), observe that at the most significant bit position (let this be m) of X, Z must have 0. But at the same time Z also have to be greater than X. Sowe must have the (m+1)th bit of Z 1 (as we want the smallest one but greater than X).Just having (m+1)th bit be 1 we made sure that Z>X. So to get the smallest number we take rest of the bit to be 0. One can easily observe that this number is actually a power of 2 immediately larger than X.

One can implement this solution in many different ways. Here is the setter's implementation: https://ideone.com/8zWcEl

Attendance Sheet

Problemsetter: Redwan Ul Haque, Mohammad Rakibul Hasan

1st

Since the students can take any letter from any part of the given string and use it, the order of the names and the order of the letters in the names do not matter.

Now, any letter can be changed into any other letter. So, as long as there are enough letters to use, everyone will be able to write their names.
If it is found that the number of letters in the given string is less than the total number of letters in all the names, then the answer is -1. Since there will never be enough letters, no matter how they are changed.
If not, then there is always an answer.
First we calculate how many times each letter is needed by iterating through each name just once. Then we find how many of each letter is already available by iterating through the given string once.
Now, we need to change the excess letters. And that's the answer.
Complexity: *O(N + L)*

2nd

We need to simply use any letter that can be used directly, and change the rest.
First we iterate over (loop over) all the names just once and count how many times each letter is needed. We can store it in an array named ***need.***
Then we iterate over the given string and if any character in the string is needed to write someone's name then we use it (decrease the count of this letter by one). Otherwise, we say it is unused and use a variable named ***unused*** to count such letters.
At the end, we count how many letters could not be used directly (were not found in the string).
We need to generate these letters by changing the unused ones.
If there are not enough unused letters to change, the answer is -1. Otherwise, the answer is the number of letters that were changed.
Complexity: *O(N+L)*

Setter's Solution: https://ideone.com/NR8e0j

Helping Out Crush

Problemsetter: Bishwajit Bhattacharjee, Mahir Shahriyar Sezan

Hint-1:
X ^ X = 0 , for any integer X. So, a number that appears even times does not contribute to the answer.

Hint-2:
A number appears in the sequence exactly NOD(x) times , where NOD(x) = number of divisors of x.

Hint-3:

Only the perfect Square numbers have odd number of divisors.(Try to convince yourself) . So, we can say that only the perfect square numbers upto n will contribute to the answer.

So, the complexity will be O(sqrt(n)).

Code:
https://pastebin.com/v39abKvs

To Infinity and Beyond
Problemsetter: Zawad Abdullah

First of all, we need to build a sieve to count how many divisors a number has.
It has the complexity **O(n * log(logn)) .** where n = $6.5 \ x \ 10^5$ as stated in the statement.

Now, we have the real problem. We cannot simply just generate the whole sequence.We need something better to answer all the $5.5 \ x \ 10^5$ queries faster.

Now, let's define a function $F(x) \ = \ the\ number\ of\ elements\ in\ the\ sequence\ not\ exceeding\ x$
Then , $if\ a \ > \ b\ then\ ,\ F(a) \ > \ F(b)\ and\ if\ a \ < \ b\ ,\ F(a) \ < \ F(b)$.

This function makes the problem a typical binary search one.
So, we take $low \ = \ 1\ and\ high \ = \ 6.5\ x\ 10^5$.
And take the $mid \ = \ (low \ + \ high) \ / \ 2$
If we see $F(mid) \ > \ k\ ,\ then\ we\ need\ a\ smaller\ answer.$
else if $F(mid) \ < \ k\ ,\ then\ we\ need\ a\ larger\ answer.$

N.B :
The use of cpp STL lower_bound() can simplify the problem to a great extent. I personally think it's worth learning .

vyrevy-array
Problemsetter: Mohammad Solaiman

The length for the Nth vyrevy-array is 2^N-1.
Let's denote the sum function for Nth vyrevy-array as S(N). The sum of the elements of Nth vyrevy-array can be pre-calculated and stored in an array in linear time.

Suppose we have the prefix sum function,
P(N, R) = sum of the elements from 1st to Rth in Nth vyrevy-array.
Then, the answer for a given query is P(N, R)-P(N, L-1).

Now, how to solve P(N, R)? We can do it recursively. Notice that, the maximum element in Nth vyrevy-array is N and it appears exactly once at 2^{N-1} th position, the mid position of the vyrevy-array.

$$P(N, R) = \begin{cases} P(N-1, R), & R < 2^{N-1} \\ S[N-1]+N, & R = 2^{N-1} \\ S[N-1]+N+P(N-1, R-2^{N-1}), & R > 2^{N-1} \end{cases}$$

Since, the state transition of the recursive function decrements N every time. The complexity will be O(N) per query.

Official solution: https://ideone.com/4LN1Xf
Alternate solution with different approach: https://ideone.com/Gre15T

Hackerman

Problemsetter: Pritom Kundu

First we make two observations. One, the order we make the moves does not matter. Two, It is pointless to make the same move twice. So, we may assume that we make each move at most once and we make the moves from a left to right manner.

We may apply a greedy strategy. We scan the first string from left to right. If we find a mismatch at index **i** we must flip each of the indices from **i** to **i+k-1.** Otherwise, we need not make a move. However, we can only do so upto index **n-k,** as after that there are not enough indices left to flip. So, we need to check if the last **k-1** values match after performing all the operations. If they don't, the answer is -1. Otherwise, the answer is the number of flips we performed.

A brute force implementation of this approach takes **O(nk)** complexity, which is not enough to pass. Instead of flipping every bit in every operation, we can only keep a flag array **fl,** where **fl[i] = 1** if the **[i...i+k-1]** has been flipped. Note that for each index, only the flags at the last **k** index can modify it. So, we may find out how many times an index

has been modified already using prefix-sum technique and find out the current value at the index. The complexity is **O(n)**.

Alternatively, we may use a segment tree or a fenwick tree to perform the updates and query, The complexity is **O(n log n)**.