**Kick Start 2019 - Round B**

# Analysis: Building Palindromes

For each query[**L**, **R**], we have to figure out if it is possible to make a palindrome from substring[**L**, **R**] of the original block string.
For any substring, we can create all possible permutations of it and check if any of those is a palindrome. For a substring of length *l*, in the worst case, there would be Factorial(*l*) permutations, and to check if a string of length *l* is palindrome is O(*l*). So the cost of each query would be O(**N***Factorial(**N**)) and thus wouldn't fit within the time limit.
One important observation is that in a palindrome, at most one character can appear odd number of times. If more than one character appears odd number of times in a string, it is impossible to rearrange that string to form a palindrome.
If all characters in a string are present even times, we can just divide the characters into two identical sets. Then we can make two strings with those two sets such that one string is the reverse of the other one. Finally concatenate them to get a palindrome.
If we have only one character let's say *x*, which is present odd number of times, we can set aside one *x*. Then we get a set of character where all characters are present even number of times, and we can construct a palindrome in somewhat similar way as described above, this time we can just put the *x* in between those two strings.

## Test set 1 (Visible)

For each query, we can count the frequencies of all characters in the substring and decide if it is possible to make a palindrome or not. The complexity of solving each query in this approach is the length of the substring, which is O(**N**). Total complexity of this approach is O(**N** × **Q**), which will be sufficient for test set 1.

## Test set 2 (Hidden)

We need to count the frequencies of each characters in a query substring, but can we make the counting of frequencies faster?
Notice that, if we calculate prefix sum of frequencies for each character on the whole input string, we can get the frequency of any character in any given substring in O(1) time.
In this approach, the time required to compute the prefix sum of the frequencies for each character is O(**N**) and hence O(**N** × |character set|) for all. And for each query substring, we can check parity of the frequency for each of the characters in O(|character set|) time. So for **Q** queries, the overall complexity of this approach is O((**N** + **Q**) × |character set|), which is sufficient for test set 2.