PREREQUISITES:

String matching (for example with KMP or hashing)

PROBLEM:

You're given a string S_0.
Create K new strings as follows:

- $S_i = S_{i-1} + rev(S_{i-1})$ for each $1 \le i \le K$

Find the number of times S_0 appears as a substring of S_K.

EXPLANATION:

Note that $S_1 = S_0 + rev(S_0)$ is a palindrome.
This means that $rev(S_1) = S_1$, so $S_2 = S_1 + S_1$ (and S_2 is also a palindrome).
Similarly, $S_3 = S_2 + S_2 = S_1 + S_1 + S_1 + S_1$.
More generally, it can be seen that for any $i \ge 1$, S_i will equal S_1 repeated 2^{i-1} times.

Since S_1 has length $2N$, this means that if S_0 occurs as a substring starting at index j, it'll also appear at all valid indices of the form $j + 2Nx$ for *integer* x.

This means it's enough to consider instances of S_0 starting at the first $2N$ indices of the string!
That is, we can do the following:

- For each $i = 1, 2, 3, \ldots, 2N$, check if the length-N substring starting at i equals S_0.
- If it does, add to the answer the number of non-negative integers x such that $i + 2Nx + N - 1 \le |S_K|$ (that is, the number of starting indices of the form $i + 2Nx$ such that there exists a length-N substring starting at it).

The first part is a rather standard string problem: we have a string (the first $3N$ characters of S_2) and a pattern (S_0), and we'd like to find all positions where the pattern appears.
This can be done in linear time in many ways: for example using hashing or the KMP algorithm.

The second part can be done with some simple math.
Recall that S_K equals S_1 repeated 2^{K-1} times.
So, for a starting index i,

- If $i \le N + 1$, this starting index will be valid in every copy of S_1, for 2^{K-1} in total.
- If $i > N + 1$, this starting index will be valid in every copy of S_1, except for the last (since there aren't enough characters to form a length-N substring).
  This gives $2^{K-1} - 1$.

So, find all valid starting indices $1 \le i \le 2N$ and add either 2^{K-1} or $2^{K-1} - 1$ to the answer for each of them, depending on their values.
Finding 2^{K-1} quickly can be done using binary exponentiation.

TIME COMPLEXITY:

$\mathrm{O}(N + \log K)$ per testcase.

PREREQUISITES:

KMP or the Z-function

PROBLEM:

You have two strings S and T. You can do the following moves:

- Choose a *proper* substring of one of the strings and copy it to either the start or end of the other; or
- Choose a *proper* substring of one of the strings and delete it.

Find the minimum number of moves needed to make both strings equal.

EXPLANATION:

It's always possible to make the strings equal in 3 moves, as we can:

- Copy the first character of S and append it to T.
- Delete all but the first character of S.
- Delete all but the last character of T.

So, we only need to check whether it's possible in < 3 moves.

Checking for 0 moves is trivial: it's only possible if $S = T$.

The discussion below assumes $|S| \geq |T|$, for convenience.

1 move

Our one move is to either delete a substring of S, or copy a substring of S to one end of T.
In fact, it turns out that the copy move is unnecessary as well!

Why?

Let $|S| = N$ and $|T| = M$.
Suppose we copy some substring of length $N - M$ from S and append it to T, after which the strings are equal.
This would mean the first M characters of S equal the first M characters of T — that is, T is a prefix of S. We could've just deleted the last $N - M$ characters of S and the strings would still be equal.

Similarly, if a substring is copied to the front of T, we'd be able to delete some prefix of S instead and achieve the same thing; so it suffices to consider only deletions.

Now, all we need to do is check if deleting some substring of S can give us T.
This is fairly easy to do: for instance, you could compute the largest common prefix of S and T (say of length x), and their largest common suffix (say of length y); both of which are easily found with a simple loop.
Then, T is obtainable by deleting something from S if and only if $x + y \geq |T|$.

2 moves

There are four possibilities to check here: the first and second move can both be deletions or copies.
However, as seen in the case of 1 move, if the last move is a copy, there will be a way to make both strings equal

with a deletion as well.
So, we only need to consider two cases: two deletions, or a copy followed by a deletion.

Time for some cases!

Two deletions

Even within this subcase, there are more subcases: we can either delete two substrings of S, or delete one substring of S and one of T.

The latter is easier to think about: if some prefix or suffix of S equals a prefix or suffix of T, we can delete things appropriately to leave only the matching strings.
Nothing more needs to be considered here.

Proof

Notice that since we aren't allowed to delete the entire string, there are two possibilities for S and T after deletion:

- Only a prefix/suffix remains; or
- Both the first and last characters remain.

If S is only a prefix, then either T's first character remains (in which case there's a matching prefix), or a suffix of T remains (in which case a prefix of S matches a suffix of T).
A similar argument holds for when only a suffix of S remains; or when only a prefix/suffix of T remains.

Finally, if both strings have something removed from their middles, their first characters should match (so there's an equal prefix anyway).

What about deleting two substrings from S? We (almost) don't need to think about it!
If the substrings we delete overlap (in terms of indices of the original string), a single deletion would've been enough; and we're only here because it wasn't.
If we delete disjoint substrings, that means the part between them lies in T — we could instead delete this part from both S and T (note that this merges the whole thing into a single large deletion in S); so we're back to the previous case!

The only exception is if the "single large deletion" is S itself, which isn't allowed.
However, note that this means the undeleted substring of S would be T itself; so we just check if T occurs as a substring of S.

All-in-all, we need to check prefix/suffix matches and if T occurs as a substring of S. All of this can be done quickly using one of the many string algorithms around: the KMP algorithm, the Z-function, you can even use hashing.

Copy then delete

Yay, more cases!
There are a few possibilities here: we can copy from S/T to the other one, and can delete from either string after copying.

Case 1**:** Copy to S, delete from S.
If the deletion is fully within the copied part, S will end up with larger length than T, so they can't be equal.
In every other case, the copy → delete step can be replaced with an equivalent delete → copy step, and as we noted earlier, we don't need to think about anything that ends with a copy.
This case gets ignored!

Case 2**:** Copy to S, delete from T.
S will have strictly larger length than T, so they'll never be equal. Ignored!

Case 3**:** Copy to T, delete from T.
Just as in case 1, if the deletion isn't fully within the copied part then an equivalent delete → copy operation exists (which we don't need to deal with).
So, the only possibility is if we copy something over from S, then delete something from within it.
In such a case, the original T must match a prefix or suffix of S; and we already checked for that anyway.
Yet again, case ignored!

Case 4**:** Copy to T, delete from S.
I'm sure you see the pattern here by now.
Notice that if deletion is done from S, the entirety of the original T remains.
So, if the copy is appended to T, then either some prefix of T will match some prefix of S, or some suffix of S will itself be T (and you get similar prefix/suffix conditions for when the copy is placed at the front).
For the final time, we've already checked these conditions earlier, so this case gets ignored, too!

tl;dr you only need to check if some prefix/suffix of S matches some prefix/suffix of T; or if T is a substring of S itself — all of which are doable with a variety of string algorithms.

If all the above checks fail, the answer is 3.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase

Prerequisites

Z-function, Prefix sums, Range Minimum Queries

Quick Explanation

For each position that Alice can take, find the maximum length interval ending at i that matches with the suffix of the string. Suppose $[i..j]$ and $[n-(j-i+1)...n-1]$ are the matching intervals, then the answer is just the minimum sub-array sum (with their right boundaries fixed) in the two intervals.

Explanation

First, let's see how to solve the following problem:
Given i, find the maximum length of a sub-string of S that ends at index i and matches with the suffix of S.

If we reverse our string S, the problem can be re-framed as:
Given i, find the maximum length of a sub-string of S that starts at index i and matches with the prefix of S.

This is a problem we can directly solve using Z-function, which can be computed in linear time. So we reverse our string, compute the z-function array (let us call it z), and finally reverse the z-function array to solve the original problem.

For a given position j where we can put Alice, we have $i = j + 1 - z[i]$
Alice will try to achieve a maximal score. This means if we put Alice in position j, she will choose a position k $(k \leq j)$ such that $Sum[k..j]$ is as maximum as possible. Let $pre[i]$ denote the prefix sum array of A. Then,

$\max_{i \leq k \leq j}(Sum[k..j]) = \max_{i \leq k \leq j}(pre[j] - pre[k-1])$
$= pre[j] - \max_{i \leq k \leq j}(-pre[k-1])$
$= pre[j] - \min_{i \leq k \leq j}(pre[k-1])$
$= pre[j] -$ RMQ($i-1, j-1$)

Where RMQ(l, r) returns the minimum value of the array $pre[]$ in the given range [l,r]. You can use Sparse Table to find the RMQ in any range in $O(1)$ time per query with $O(NogN)$ preprocessing.
For Bob, we can find his maximum score by finding the same as above in the interval $[n - z[i]...n-1]$ and take the maximum of the two, giving us the maximum team score if we put Alice at j.

The answer is just the maximum of this value, taken for all j.

Alternate approaches

The maximum matching can be found using polynomial hashing and binary search as well. For RMQ, using a segment tree alternatively also fits in the constraints.

PREREQUISITES:

Ad-hoc, String

PROBLEM:

Given a binary string, find if it is possible to make all its digits equal by flipping exactly one digit.

QUICK EXPLANATION:

All the digits of the binary string can be made equal by flipping exactly one digit if and only if the original string has exactly one digit equal to zero, or exactly one digit equal to one.

EXPLANATION:

Suppose that the input string has exactly one digit equal to one, and all other digits are zero, then we can flip this digit, which will result in a string with all digits equal to zero. Similarly, if the input string has exactly one digit equal to zero, and all other digits are one, then flipping this digit would result in a string with all digits equal to one.

On the other hand, if all digits of a string can be made identical by doing exactly one flip, that means the string has all its digits equal to one another except this one digit which has to be flipped, and this digit must be different than all other digits of the string. The value of this digit could be either zero or one. Hence, this string will either have exactly one digit equal to zero, and all other digits equal to one, or exactly one digit equal to one, and all other digit equal to zero.

Therefore, we only need to check whether the string has exactly one digit equal to zero/one, and if so, the answer is yes; otherwise the answer is no.

```
void f(string str) {
  int num_zeros = 0;
  int num_ones = 0;

  for (char ch : str) {
    if (ch == '0') {
      ++num_zeros;
    } else {
       ++num_ones;
    }
  }

  if (num_zeros == 1 || num_ones == 1) {
    print("Yes\n");
  } else {
    print("No\n");
  }
}
```

TIME COMPLEXITY:

O (N), where N is the length of the string.

PROBLEM:

You have a string S, K strings $P_1, \ldots, P_K$, and some characters given in a frequency array C.

For each x from 0 to K, find out whether it's possible to make S greater than exactly x of the given strings.

EXPLANATION:

One immediate observation we can make is that appending characters to S will never make it any smaller lexicographically: it can only make it larger.

Let's try to solve for a specific value of x first.

To make things easier for us, first sort the strings P_i in increasing order, i.e, $P_1 \le P_2 \le \ldots \le P_K$.
Now, notice that making S greater than exactly x strings means we want to append some characters to S to make it satisfy $P_x < S \le P_{x+1}$.
In particular, it's enough to only look at the two strings P_x and P_{x+1}, so let's do that.

From here, we can do a bit of basic casework to throw out a few simple cases, each time making what S we have to deal with more specific.

- If $S > P_{x+1}$, obviously the answer is `No`.
  - Now we have $S \le P_{x+1}$.
- If $P_x = P_{x+1}$, again the answer is `No`.
  - Now we have $S < P_{x+1}$.
- If $S > P_x$, we don't need to append anything: we're already done and the answer is `Yes`.
  - Now we have $S \le P_x < P_{x+1}$.
- If the length of S is greater than the length of P_x, then the answer is `No`.
  - Now we further have $|S| \le |P_x|$.
- If S is not a prefix of P_x, once again the answer is `No`.
  - Now S is a prefix of P_x.

This is the end of the 'simple' cases, which don't depend on the extra characters in C at all.
From here on, we have to check whether appending some characters to S can make it $> P_x$.

This check can be done greedily position-by-position, starting from $i = N + 1$.
When we are at position i,

- If S is a prefix of both P_x and P_{x+1}, and we can append some character that is $> P_{x,i}$ but $\le P_{x+1,i}$, we are immediately done and the answer is `Yes`.
- If S is not a prefix of P_{x+1}, then simply check if we can append *any* character $> P_{x,i}$: if we can, the answer is `Yes`.
- If the above cases are not possible, note that our only hope of making $S > P_x$ in the future is to append exactly $P_{x,i}$ and maintain S as a prefix of P_x, so check if this is possible.
  - If it is possible, do so. Make sure to update whether S is still a prefix of P_{x+1} or not.
  - If it is not possible, the answer is immediately `No`.
- A little bit of care needs to be taken when i is larger than the length of P_x and/or P_{x+1}: these need to be special-cased.

Note that this algorithm runs in something like $\mathrm{O}(26 \cdot (|P_x| + |P_{x+1}|))$ time.

So, simply running this algorithm for every x is already fast enough! Every string is processed twice this way, giving us a solution in $\mathrm{O}(26 \sum |P_i|)$ which is good enough.

Depending on your implementation, you might have to take special care of $x = 0$ and $x = K$, since the strings P_0 and P_{K+1} don't exist.

TIME COMPLEXITY

$\mathrm{O}(N + K + 26 \sum |P_i|)$ per test case.

PREREQUISITES:

Palindromic tree/Manacher's algorithm (or hashing), string suffix structures (or hashing), binary search

PROBLEM:

You have a string S. Form a new string T by taking all the distinct palindromic substrings of S, sorting them lexicographically, and then concatenating them in this order into a single large string.

Now you have Q queries, each of the form L, R. Given a query, you need to print the substring of T that starts at position L and ends at position R, or say that such a substring doesn't exist (i.e R is larger than the size of T)

EXPLANATION:

The solution can be broken into 3 parts: finding all the distinct palindromes of S, sorting them, and finally answering queries.

Finding palindromes

It so happens that there is a data structure that finds all distinct palindromic substrings of a string and keeps information about them in linear time, called the palindromic tree. You can read about it here or here 6 .

However, knowledge of this data structure isn't strictly needed to solve the problem: you can apply some other well-known algorithms as well.

The most important property of palindromic substrings is that there are at most N of them.

Proof

This can be proved by induction on the length of the string.
When $N = 1$, clearly there is only one palindromic substring.

Now, consider what happens when you append a character to a string of length N to form one of length $N + 1$ how many new palindromes are created?
Clearly any new palindrome must end at the last character, so consider all palindromes ending at the last character. Only the largest one of these is a potentially new palindrome: all the shorter ones will also appear as a prefix of the longest one (since they form a suffix of it), and thus already exist within the first N characters.

Adding a new character adds at most one new palindromic substring. So, a string with N characters has at most N distinct palindromic substrings.

The above proof also gives us an idea to compute these N substrings: each of them must be the longest palindrome ending at some index of the string, so we just need to be able to compute this.

This can be done with the help of Manacher's algorithm.
Manacher computes, for each index i, the (half-length of the) longest palindrome centered at i. Suppose this length is k. Then, the longest palindrome ending at $i + j$ for each $i \le i + j \le i + k$ has length at least $j + 1$.
There are $2N - 1$ restrictions of this form (N from the odd lengths, $N - 1$ from the even lengths). Then, the actual length of the longest palindrome ending at each position can be computed from this using a sweep line.

Finding these lengths can also be done with a combination of hashing and binary search. You can read about it here 2 .

This part takes $\mathrm{O}(N)$ or $\mathrm{O}(N \log N)$, depending on implementation and method chosen.

Sorting the palindromes

Now that we have found our $\leq N$ distinct palindromes, we need to sort them lexicographically.

This is a more general problem of sorting several substrings of a string, and can be accomplished in several ways:

- Build the suffix array of the string, which automatically gives us this information; or
- Once again, use hashing and binary search — this time, to compare two substrings in $\mathrm{O}(\log N)$ time and use a standard sorting algorithm for an $\mathrm{O}(N \log^2 N)$ implementation. You can read about the details of this here 2 .

Depending on the method chosen, this step is $\mathrm{O}(N), \mathrm{O}(N \log N)$, or $\mathrm{O}(N \log^2 N)$.

Also note that depending on implementation, you might have duplicate palindromes at this stage. After sorting them, make sure to remove duplicates or it'll mess up the next step.

Answering queries

Now that the substrings have been sorted, answering queries is fairly easy.
Let the length of the i-th palindrome be len_i.
Given a query $[L, R]$:

- if R is larger then the sum of all len_i, print -1
- Otherwise, find the first index i such that $len_1 + len_2 + \ldots + len_i \geq L$. This can be done with binary search on the prefix sums of len_i.
  Position L of T is then in the i-th substring, and its index can be found easily.
- Now, simply start printing characters from L till you reach R, each time moving to the next palindrome when you reach the end of one.

The complexity of answering a single query is $\mathrm{O}(\log N + len(output))$. Since we are guaranteed that the sum of output lengths across all queries is bounded, this is good enough.

TIME COMPLEXITY

$\mathrm{O}((N + Q) \log N)$ per test case (with maybe an extra $\log N$ if hashing is used).

PROBLEM:

You are given a binary string S of length N. You can perform the following operation on S:

Pick any set of indices such that no two picked indices are adjacent.
Flip the values at the picked indices (i.e. change 0 to 1 and 1 to 0).

For example, consider the string S = 1101101.
Pick the indices $\{1, 3, 6\}$.
Flipping the values at picked indices, we get $\underline{1}1\underline{0}11\underline{0}1 \rightarrow 0111111$.
Note that we cannot pick the set $\{2, 3, 5\}$ since 2 and 3 are adjacent indices.

Find the **minimum** number of operations required to convert **all** the characters of S to 0.

EXPLANATION:

For each test case we have a binary string of length N.

Now as *no adjacent bits can be flipped together*. Three cases are possible:

- When the string is already 0, then the number of minimum operations will be **zero**.
- When **no two adjacent bits are set**. In this case all the set bits can be converted to 0 in 1 operation.
- When the string contains adjacent set bits. In this case it can be shown that the answer will be 2 since all the odd-indexed set bits and even-indexed set bits can be identified as **two different sets**.

Evaluating all the above possibilities gives us the result.

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

A bracket sequence S is called *dense* if one of the following is true:

- S is empty.
- $S = (X)$ where X is dense.

You are given a bracket sequence S. What is the minimum number of brackets you must remove to make it dense?

EXPLANATION:

In order to solve this problem we can keep a count of all the opening brackets and closing brackets for each position in the string. First we will calculate the number of closing brackets in the string. This would be denoted by $close$

The maximum answer can be n as we would get a dense string if we remove all the brackets. Now we traverse from 1 to n. Initially we would have 0 opening brackets(denoted by $open$) and $close$ closing brackets. At each position there can be two cases:

- $s[i] = ($: Here we increment $open$ by 1. Thus we have $open$ opening brackets in the left and $close$ closing brackets at the right, so the maximum length of dense string that can be formed at this position is $min(2 \times open, 2 \times close)$. Thus,

$$ans = min(ans, n - min(2 \times open, 2 \times close))$$

- $s[i] =)$: Here we simply decrement $close$ by 1.

By the end of our traversal we will have our answer.

TIME COMPLEXITY:

$O(N)$, for each test case.

PROBLEM:

Chef considers a string consisting of lowercase English alphabet *beautiful* if **all** the characters of the string are **vowels**.

Chef has a string S consisting of lowercase English alphabet, of length N. He wants to convert S into a *beautiful* string T. In order to do so, Chef does the following operation on **every** character of the string:

- If the character is a **consonant**, change the character to its **closest vowel**.
- If the character is a **vowel**, don't change it.

Chef realizes that the final string T is not unique. Chef wonders, what is the total number of **distinct** *beautiful* strings T that can be obtained by performing the given operations on the string S.

Since the answer can be huge, print it modulo $10^9 + 7$.

Note:

- There are 26 characters in the English alphabet. Five of these characters are vowels: `a`, `e`, `i`, `o`, and `u`. The remaining 21 characters are consonants.
- The closest vowel to a consonant is the vowel that is at least distant from that consonant. For example, the distance between the characters `d` and `e` is 1 while the distance between the characters `d` and `a` is 3.
- The distance between the characters `z` and `a` is 25 and **not** 1.

EXPLANATION:

In this problem we need to change all the consonants to their nearest vowel. The consonants that have a unique vowel nearest to it will just have one choice, that is to change into that vowel, however for vowels that have 2 vowels at same closest distance, it will have 2 choices. Thus we just need to calculate all consonants that have 2 choices. Let us denote this by $count$. Then our final answer would be:

$$answer = 2^{count}$$

TIME COMPLEXITY:

$O(N)$, for each test case.

PROBLEM:

For a binary string A, let $f(A)$ denote its *badness*, defined to be the difference between the number of zeros and number of ones present in it. That is, if the string has c_0 zeros and c_1 ones, its badness is $|c_0 - c_1|$. For example, the badness of "01" is $|1-1| = 0$, the badness of "100" is $|2-1| = 1$, the badness of "1101" is $|1-3| = 2$, and the badness of the empty string is $|0-0| = 0$.

You are given an integer N and a binary string S.

You would like to partition S into K disjoint subsequences (some of which may be empty), such that the maximum badness among all these K subsequences is **minimized**. Find this value.

Formally,

- Let $S_1, S_2, \ldots, S_K$ be a partition of S into disjoint subsequences. Every character of S must appear in one of the S_i. Some of the S_i may be empty.
- Then, find the *minimum* value of $\max(f(S_1), f(S_2), \ldots, f(S_K))$ across all possible choices of $S_1, S_2, \ldots, S_K$ satisfying the first condition.

EXPLANATION:

Let there be O ones and Z zeros in S. After dividing S into K boxes let there be Z_i zeros and O_i ones in box i.

We can write, $Z - O = (Z_1 - O_1) + (Z_1 - O_1) + ...(Z_K - O_K)$.

If $Z > O$ then it is always better to divide S so that for each of the K boxes $Z_i \geq O_i$. This is because if for some box x we have $Z_x < O_x$, there has to be at least one box y having $Z_y > O_y$ because $Z > O$ overall. Reducing 1 zero from box x and placing it in box y instead would have helped to reduce the badness for both boxes x and y by 1. This is clearly not optimal. Using the above observation it is clear that $Z_i - O_i$ is the badness for box i. Hence, we need to minimize the maximum among $(Z_i - O_i)$ and this is achieved when all the K terms have as close values as possible and add up to $Z - O$ which is known. The maximum term after such division is $\lceil \frac{Z-O}{K} \rceil$, which is our answer.

For $Z < O$ the vice versa (replace ones by zeros and zeros by ones) is true and the answer is $\lceil \frac{O-Z}{K} \rceil$ and if $Z = O$ then the answer is 0 because we can place the entire S into 1 box.

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

Chef has a string S consisting only of English lowercase letters (a - z). However, Hitesh does not like it and wants it to be reversed.

Hitesh wonders what is the minimum number of operations required to reverse the string S using the following operation:

- Select any i such that $1 \leq i \leq |S|$ and remove S_i from its original position and append it to the end of the string (i.e. shift any character to the end of the string).

For e.g. if $S = \texttt{abcde}$ and we apply operation on $i = 2$, then S becomes $\texttt{acdeb}$.

Help Hitesh find the minimum number of operations required to reverse S.

It is guaranteed that it is possible to reverse the string in a finite number of operations.

QUICK EXPLANATION:

- In the optimal scenario, we will never remove the same character twice. In this sentence, we are treating each index element as a distinct character. So, in aba, the first and the third a are distinct for this sentence.
- Let S' denote the set of indices (in the original string) that we have removed and appended at the end, and let S denote the set of indices that we have not operated on. How will the string look after all the operations.
- Make the cardinality of S as large as possible.

EXPLANATION:

The first observation is in the optimal scenario, we should never remove the same character twice. To see this, consider a sequence of operations O_1 in which the same character c is removed twice. Consider another sequence of operations O_2, where the first operation in which the character c was removes, is removed from O_1, and rest everything remains same. We can see that both O_1 and O_2 will lead to same string, with $|O_2| < |O_1|$.

Let $S = \{i_1, i_2, ..., i_k\}$ be the set of indices which have not been operated on. Let us call the new string as new_str. We have:

$str = s_1 s_2 ... s_n$
$rev_str = s_n s_{n-1} ... s_2 s_1$
$new_str = s_{i_1} s_{i_2} ... s_{i_k}\ s_{i'_1} ...,$

Because $rev_str = new_str$, we have $s_n = s_{i_1}, s_{n-1} = s_{i_2}$, and so on. So to minimize the size of set S', we should maximize the size of set S, which is equal to k. In other words, we need to find out the longest prefix of rev_str that appears as a subsequence in str.

We can find this prefix greedily, using the algorithm described here in $O(N)$ time.

TIME COMPLEXITY:

$O(N)$ per test case.

PREREQUISITES

Strings, Greedy

PROBLEM

You are given a binary string S of length N. You can reverse any substring of S in one operation, find the minimum number of operations required to sort S.

EXPLANATION

Hint

The resulting string should be such that all '0's appear consecutively before a '1' appears.

Detailed Explanation

To sort S, we can iterate through the string and reverse every leftmost instance of "$1, 1, ..., 1, 0, ..., 0, 0$", i.e. a substring of S with consecutive '1's and then consecutive '0's. This is guaranteed to sort S in the minimum number of operations.

Intuition for correctness

While iterating through S, if you reverse an instance of "$1, 1, ..., 1, 0, ..., 0, 0$" then you would obtain a prefix of '0's, as we are essentially *pushing forward* the occurrences of '1's that are present before a '0'. So, the resulting string would have a prefix of '0's and a suffix of '1's, and hence would be sorted in the minimum number of operations.

This simply reduces to counting the number of occurrences of "10" that appear in the string and printing it. This is because the number of "$1, 1, ..., 1, 0, ..., 0, 0$"s that appear in the string are the same as the number of "10"s that appear in the string.

TIME COMPLEXITY

The time complexity is $O(|S|)$ per test case.

PROBLEM:

A string S is called *good* if none of the substrings of S of length ≥ 2 are palindromic.

For e.g. $S = \text{hello}$ is not good but $S = \text{world}$ is good.

You are given a string A of length N (containing lowercase Latin letters only). Rearrange A to form a good string or determine it is not possible to do so.

Recall that a string is called palindromic if it reads the same backwards and forwards. For example, noon and level are palindromic strings.

EXPLANATION:

Observation: If in a string there are no palindromic sub-strings of length 2 or 3, then there are no palindromic sub-strings in the string(except for sub-strings of length 1).

Keeping this observation in mind, we divide the entire string into three containers c_1, c_2, c_3 where, $c_i = \{j,\ such\ that\ j\ mod\ 3 = i, 1 \leq j \leq n\}$. In order to avoid palindromic sub-strings of length 3, we must keep a distance of at least 2 between each same character or we must keep them in same containers. Thus for each character we have a limit of its frequency as:

$$limit = \lceil \frac{n}{3} \rceil$$

If frequency of any character exceeds this limit then it won't be possible to make it a good string.

Now let's talk about the case when there are multiple characters that have frequency equal to limit. Taking another variable $afford$ as $(n\ mod\ 3)$.

- $afford = 0$: All containers are of length $\lceil \frac{n}{3} \rceil$.
- $afford = 1$: Only one container is of length $\lceil \frac{n}{3} \rceil$.
- $afford = 2$: Two containers are of length $\lceil \frac{n}{3} \rceil$.

Let us define bad as number of characters having frequency equal to $limit$. If bad is more than the number of containers having length equal to $\lceil \frac{n}{3} \rceil$, then it won't be possible to construct a good string.

If none of such cases arise then we can confidently say that we can create a good string.

Now we loop through each characters in descending order of their frequency and if its frequency is equal to $limit$, then we fill it in all the positions of container whose size is equal to that of $limit$ else we can fill its characters in the containers in the order $c_3 -> c_2 -> c_1$

TIME COMPLEXITY:

$O(Nlog(N))$, for each test case.

PREREQUISITES:

Strings, Binary Search

PROBLEM:

Chef likes to spend his summer vacation playing with string S. But he only likes the characters a and b to be part of his string.

Initially the string S is empty and Chef will build the string into his favourite alternate pattern in the following way.

- Add a once to string S.
- Add b twice to string S.
- Add a thrice to string S.
- and so on infinitely.

Now Chef will crop this infinitely large string to a length N and calculate the beauty of the string S.

The beauty of a string S is the sum of values of all substrings of S and the value of string S is the number of times the character changes from a to b or vice versa in the string.

Can you help Chef find the beauty of his string S of length N.
Since this calculated number can be large, output it modulo 1000000007.

EXPLANATION:

We can see if we take the first two characters then there would be 1 change and number of substrings that includes the first two characters are $N-1$.
Now we move on to the $2nd$ change that occurs at 3_{rd} position for first 4 characters of the string and number of substrings that includes this change would be $3 \times (N-3)$.
Similarly the number of substrings that includes the 3_{rd} change would be $6 \times (N-6)$.
So basically our answer would be something like:

$$\begin{aligned} ans &= 1 \times (N-1) + 3 \times (N-3).....k \times (N-k) \\ &= N \times (1+3+.....+k) - (1^2+3^2....+k^2) \end{aligned}$$

We can easily calculate k using binary search and pre-calculate the sum of the series to get out final answer.

TIME COMPLEXITY:

$O(log(N))$, for each test case.

PROBLEM:

Given a string $s[1 \sim n]$, does there exist a permutation p of $1, 2, \ldots, n$ such that, if we let $t[i] = s[p[i]]$, then t is palindrome?

EXPLANATION:

This problem is equivalent to: can we rearrange the characters of s into a permutation?

Let's count the number of occurrences of every characters in a-z. For a char c, let $cnt[c]$ be the number of times c occur in s.

If there are two chars $c_1, c_2 (c_1 \neq c_2)$ such that both $cnt[c_1]$ and $cnt[c_2]$ is odd, then the answer is no.

- To see this, consider any palindromic string t and any char c. Suppose $t[i] = c$.
- If the length n of t is even, then $t[n - i + 1] = c$. Since n is even, $n - i + 1$ and i has different parity and hence $i \neq n - i + 1$. Also $n - (n - i + 1) + 1 = i$, so we say i and $n - i + 1$ is a pair. Any pair must use the same character, so any character must occur an even number of times.
- If the length n of t is odd, the above argument holds, too, except that the central char($t[\frac{n+1}{2}]$) is matched with itself. In this case, this char appears an odd number of times, and is the only char that appears an odd number of times.
- See figure below. $n = 7$, pairs are $(1, 7), (2, 6), (3, 5)$, and $t[4]$ is matched with itself, and hence appears an odd number of times.

- In conclusion, if C is the number of chars c that $cnt[c]$ is odd, and s can be arranged into a palindrome, then $C \leq 1$.

The converse is also true: if there are at most 1 characters c with $cnt[c]$ odd, then the answer is yes. Here is **one of** the constructions.

- Let's construct the permutation char-by-char. Let c iterate from a to z.
  - If $cnt[c]$ is odd, we ignore c(that is, we consider c later). At most one such c exists.
  - We maintain two pointers l and r. Initially $l = 0, r = n + 1$. The permutation $p[1 \sim l]$ and $p[r \sim n]$ is already filled.
  - We scan s. Each time we meet a c, say $t[i] = c$,

1. if this is the $1, 3, 5, \ldots$,(odd) time we meet c, we assign it to the left: $l \leftarrow l + 1$ and $p[l] \leftarrow i$;
2. otherwise (this is the $2, 4, 6, \ldots$ time we meet c), we assign it to the right: $r \leftarrow r - 1$ and $p[r] \leftarrow i$.
3. The picture below shows how we fill p when $s =$abdacbabdccba, $i =$b.

- At last, in case that n is odd, we need to deal with the c where $cnt[c]$ is odd. That's easy: $p[l + 1 \sim r - 1]$ is unfilled, and we fill them by char c.
- This is a valid answer. Time complexity: $O(\sigma \cdot n)$. ($\sigma = 26$)

ALTERNATIVE SOLUTIONS

This is a constructive problem. I think there'll be many solutions. **Please feel free to share your solution.**

PROBLEM:

Chef wants to implement wildcard pattern matching supporting only the wildcard '?'. The wildcard character '?' can be substituted by any single lower case English letter for matching. He has two strings X and Y of equal length, made up of lower case letters and the character '?'. He wants to know whether the strings X and Y can be matched or not.

EXPLANATION:

==========
We can reduce problem of matching two strings X and Y to matching individual characters for each index $0 \leq i < N$. If all characters can be matched, then we can say that both strings can also be matched.

MATCHING A CHARACTER

We need to check if two characters a and b can be matched or not. If either of them is '?', then we can always match them by filling it with the required value. If both are '?', still we can give any same value to both of them.

If both are not '?', then we just need to check if the current values are same or not.

For implementation, see setter's commented code.

PROBLEM:

Given integers N and X, generate a palindrome of length N consisting of **lowercase English alphabets** such that the number of **distinct** characters in the palindrome is **exactly** X.
If it is impossible to do so, print -1 instead.

EXPLANATION:

Lets tackle this problem case by case:

- Case 1: When $N = 1$:
  Here the answer can be any character, example ‘a’.
- Case 2: When $N < (2 \times X) - 1$:
  Here it won’t be possible to construct the required string, since for X distinct characters we need to use at least $(2 \times X - 1)$ characters.
- Case 3: When $N = (2 \times X) - 1$:
  Assuming s to be a string of length X having all distinct characters, we can construct the following string as our answer:

$$s_1 s_2 \ldots s_{x-1} s_x s_{x-1} s_{x-2} \ldots . s_1$$

- Case 4: When $N > (2 \times X) - 1$:
  Assuming s to be a string of length X having all distinct characters, we can have three parts as:

$$left = s$$
$$mid = A\ string\ of\ length\ (N - 2 \times X)\ having\ all\ characters\ as\ s_1$$
$$right = reverse(s)$$

  Thus our final answer for this case would then be:

$$ans = left + mid + right$$

TIME COMPLEXITY:

$O(N)$, for each test case.

PRE-REQUISITES:

Strings, Arrays, ASCII system for characters 1

PROBLEM:

Given a string S on N characters, we have to perform the following operations-

- swap adjacent characters. Eg- Swap 1 with 2, 3 with 4 and so on.
- swap every alphabet OF THE ORIGINAL STRING with its mirror reverse, i.e. if ASCII value of the character is K , swap it with character of ASCII value $'a' + ('z' - K)$. Hence, $'a'$ is replaced by $'z'$, $'b'$ with $'y'$ and so on.

QUICK EXPLANATION:

Key to AC- This is a majorly implementation question. Proper practice in these topics is the key to quick AC.

Swapping of the adjacent characters is trivial. The replacement of alphabets can be easily done by realizing that every alphabet is replaced by $'a' + ('z' - K)$ counterpart, where K is ASCII value of the character.

EXPLANATION:

This editorial will have only a single section, as its majorly an easy implementation problem. We will be discussing setter's solution owing to its neat implementation.

1.Setter's solution-

The first thing to carry out is the swap operation. Remember the kind of swapping. Let me denote character at index i with c_i. We have to swap c_0 with c_1, c_2 with c_3 and so on. Familiarity with arrays and strings is needed. You can use a temporary variable for swapping, or perhaps put std::swap 4 function from C++ STL. Both the ways are given in tab below-

Click to view

1. Using std::swap-

```
for(int i=0;i< n-1;i+=2){
            swap(s[i],s[i+1]);
        }
```

2. Using temporary variable-

```
for(int i=0;i< n-1;i+=2){
            char c=s[i];
            s[i] = s[i+1];
            s[i+1] = c;
        }
```

Now all that remains is to do the alphabet replacing. We can do that trivially using the formula given above. An example is given under tab. We will replace $s[i]$ with $s[i] =' a' + ('z' - s[i])$;

Click to view

```
for(int i=0;i< n;i++){
            s[i]  = 'z' + 'a'- s[i];
```

```
        }
```

SOLUTION:

The codes of setter and tester are pasted below in tabs for you guys to refer if the links dont work. @admin needs some time for linking solutions, hence code was pasted for your convenience.

Setter 56

Click to view

Tester 44

Click to view

$Time\ Complexity = O(N)$

CHEF VIJJU'S CORNER 😃

1. Some contestants wrote a logic similar to below while coding for swapping part-

```
for(int i=0;i< n-1;i++){
            char c=s[i];
            s[i] = s[i+1];
            s[i+1] = c;
        }
```

Is something wrong there? :o

2. This time, we were able to derive a formula. What if, this wouldnt be possible? What if mapping would had been random, eg - Replace 'a' with 'g', 'h' with 'i', 'b' with also 'g' , or perhaps what if mapping would be given in input? Its easy still! Just make an array of size 26 and store which alphabet is $i^{'}th$ one mapped to. Meaning, $table[i]$ would store the character to which alphabet with ASCII $^{'}a^{'} + i$ gets mapped to. Now, another version, what if I say that the characters are upto 10^9, but only 10^5 are used and need to be mapped? I want to bring std::map 2 data structure to your attention. 🙂

PROBLEM:

You are given a binary string A of length N.

You can perform the following type of operation on the string A:

- Choose two **different** indices i and j $(1 \le i, j \le N)$;
- Change A_i and A_j to $A_i \bigoplus A_j$. Here $\bigoplus$ represents the bitwise XOR operation.

Find the **minimum** number of operations required to convert the given string into a palindrome.

EXPLANATION:

A **palindrome** is a string that reads the same forwards and backwards. If N is the length of the string A, then for each i from 1 to $ceil(N/2)$, $S_i = S_{N-i+1}$
Let the count of indices i from $i = 1$ to $i = ceil(N/2)$ such that $S_i! = S_{N-i+1}$ be equal to cnt, then the answer is $ceil(cnt/2)$ as each operation can decrease the count of such indices by at most 2.
Decrease count of such indices by 2
Let $S_{j_1}! = S_{N+1-j_1}$ this means that exactly one of these two values is 1, pick another similar pair where $S_{j_2}! = S_{N+1-j_2}$. Take the $bitwise\ XOR$ of the two (out of these four) values which are equal to 1 and set each of them to 0.
Decrease count of such indices by 1
Let $S_{j_1}! = S_{N+1-j_1}$ this means that exactly one of these two values is 1. Take the $bitwise\ XOR$ of these two values (S_{j_1}, S_{N+1-j_1}) and set each of them to 1.

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

A string consisting only of parentheses (and) is called a parentheses string. The balanced parentheses strings are defined recursively as follows:

- An empty string is a balanced parentheses string.
- If s is a balanced parentheses string, then (s) is also a balanced parentheses string.
- If s and t are balanced parentheses strings, then st (concatenation of s and t) is also a balanced parentheses string.

For a parentheses string, an operation on this string is defined as follows:

- Choose any substring of this string which is balanced.
- Remove this substring and concatenate the remaining parts.

Given a parentheses string S, find the **minimum** length of the string one can get by applying the above operation any number of times. Also find the **minimum** number of operations required to achieve this minimum length.

EXPLANATION:

The first observation that we can make is that on removing a balanced parenthesis from the string, it won't affect other balanced parenthesis in the remaining string. So what we can do is remove all the balanced parenthesis from the string and then on the final string left, we can calculate the number of operations by counting the number of missing segments in it.

In order to implement it we can replace the string by a vector v, where

$$v[i] = \begin{cases} (i+1), & if\ s[i] = (\\ -(i+1), & if\ s[i] =) \end{cases}$$

Now we can take another vector, say res and loop through the vector v and for any index i, which has a closing bracket, i.e $v[i] < 0$, if we can find a corresponding opening bracket at index j, that makes string $s[j...i]$, balanced, i.e we find res not empty and last element of res to be positive, we simply remove the last element of res, implying that substring $s[j....i]$ is removed, otherwise we simply add $v[i]$ to res

Thus in the end res would contain the indexes of the final string after removing all the possible balanced parenthesis from the original string.
The size of res would be the first part of the answer, i.e the length of the final string. As for the number of operations, we have the indexes of the characters of the final string, so we can just count the number of missing segments to get the second part of the answer.

TIME COMPLEXITY:

$O(N)$, for each test case.

PROBLEM:

Gotham City is the home of Batman, and Batman likes to keep his people together.

There are N houses in the city. The resident of the ith house is of type A_i. It is known that people of the same type are friends and people of different types are not friends.

To maintain peace, Batman can do the following type of operation:

- Choose indices i and j $(1 \leq i \leq j \leq N)$ and reverse the substring $A[i,j]$.

Determine the length of the **longest** range of friends after applying **at most** k operations on the string for all $0 \leq k \leq N$.

EXPLANATION:

First let's see how we can increase the length of substrings having same character by the reversal operation as given in the question.
For a string say, s and for character say, x, we first take the longest and 2nd longest substrings having all characters as x. We are taking the longest and 2nd longest since we want to maximize the length of the substring formed in the given operation. Say they are at position i_1 and i_2 with length l_1 and l_2. Assuming i_1 is less than i_2, we can just reverse the substring $s[i_1....i_2 - 1]$ to get substring of length $(l_1 + l_2)$ having all x. We can keep repeating this operation till we get the longest substring that contains all the x that are present in string s. Also we would observe that the number of operations required to get the above longest substring would be the number of substrings of x in the original string which would be always less than n, i.e the length of the string s.

Now coming back to this question, what we can do is repeat the operations as defined in the above paragraph for each character from a to z and for each operation take the length of the maximum possible substring that can be formed among a to z as our answer for that operation.

TIME COMPLEXITY:

$O(NlogN)$, for each test case.

PROBLEM:

Chef recorded a video explaining his favorite recipe. However, the size of the video is too large to upload on the internet. He wants to compress the video so that it has the minimum size possible.

Chef's video has N frames initially. The value of the ith frame is A_i. Chef decides to remove a frame only if its value is **equal** to the value of **either** of its neighbors.

Find the **minimum** number of frames Chef can achieve.

EXPLANATION:

Basically in this problem we need to calculate the total number of substrings having same characters. Here the video is considered as a string with each frame as individual character.

Initially we start with $n\,frames$. On traversing the string whenever we find two adjacent frames as equal, we can decrement our $frames$ by 1, since those two frames would belong to the same substring.
After traversing we would get the final answer as $frames$.

PREREQUISITES:

Brute force, Strings

PROBLEM:

Lately, Chef has been inspired by the **pawri** meme. Therefore, he decided to take a string S and change each of its substrings that spell **party** to **pawri**. Find the resulting string.

QUICK EXPLANATION:

We will traverse the given string and at any instant we find the word **party** we shall replace it with **pawri**.

EXPLANATION:

We need to replace all the occurrences spelling **party** with **pawri**. To achieve this we will run a loop from the beginning of the string to 4 less than its end and at every index check for the presence of the substring **party** starting from that index. In other words, for every i such that $0 \le i < S.size() - 4$, we check whether $S[i : i + 4]$ spells **party** and if it does, we change $S[i : i + 4]$ to **pawri** instead. We needn't go beyond 4 less than the length of the string because any substring starting at indices beyond that would have length less than 5 (which is the desired length to match with **party**). Also note that once we have found a substring matching **party** say from index i to $i + 4$, we can skip checking substrings starting from $i + 1$ to $i + 4$ because no suffix of **party** is also its prefix, and thus none of the substrings starting from any of those characters would turn out to be **party**.

The substring $S[i : i + 4]$, on arriving at subsequent indices can be obtained (for matching) as well as replaced using functions defined by respective languages that serve the purpose (for example `S.substr(i, length)` in C++ for returning substring starting at index i of specified length and `S.replace(i, length, Sr)` for replacing it with the string Sr). In this particular case, as the substring is small in length, individual characters can be matched and replaced easily from i to $i + 4$ without the aid of such functions as well.

Another way to approach the problem could utilize [KMP algorithm for pattern searching]. But it would not significantly affect the time complexity, as no two consecutive windows will both be prefixes to the pattern. This happens because our pattern "party" consists of all different characters.

TIME COMPLEXITY:

$O(N)$ per test case.
N being the length of the string given, $T(N) = (N - 4) \times 5$.
Note that `std::string::substr` as [documented] has its complexity unspecified, but generally linear in the length of the returned object, which in this case is 5.

PROBLEM:

Chef and his wife Chevy are alchemists in search of the legendary Philosopher's Stone. The Philosopher's Stone is a very powerful substance which is believed to have the power to transmute precious metals, stones and even lives of living beings. Chef wishes to use the stone to restore his son Jim's body to its original form following a terrifying accident in their lab.

It is known that Philosopher's Stones break apart into an odd number of shards when they're used up. After searching far and wide, Chef has collected N shards of what he believes to be a previously-used Philosopher's Stone, where N is an odd integer. He believes that fusing these shards back together will yield a real stone in its entirety.

Leveraging his encyclopaedic knowledge of alchemy, Chef recalls an ancient book with the following ritual for fusing shards:

1.Arrange the N shards in a row, numbered from 1 to N.

2.Note the colour Ci of each shard i, which is either auburn or black (denoted by the characters A and B respectively).

3.Choose a consecutive sequence of three shards, remove them from the row, and fuse them together into one new shard. The three shards must not all be the same colour, or such an intense concentration of similar energies will cause a dangerous rebound explosion. Instead, there must be two differently-coloured shards that will neutralise each other, get converted into energy (via the Law of Equivalent Exchange), and become infused with the third shard. In other words, assuming no explosion, the resulting shard's colour will be equal to the most common colour present amongst the three chosen shards.

4.Place the resulting shard back into the original gap in the row, with the remaining shards in the same relative order before and after it.

5.Repeat steps 3 and 4 until only one shard remains.

Note that each time steps 3 and 4 are performed, the number of remaining shards decreases by 2. Therefore, given that the initial number of shards is odd, enough successful repetitions will result in exactly one final shard: the Philosopher's Stone. Help Chef determine whether or not it's possible to get to that point and hopefully save his son Jim, without causing any rebound explosions along the way.

EXPLANATION:

You have to just arrange data in such a way that no A should come next and before to B in pair of three… it simply mean that the count of A should be equal to count of B+1 or count of B is equal to count of A+1. if this condition satisfied you will print Y otherwise N.

PROBLEM:

You are given two integers X and Y. You need to construct two different strings S_1 and S_2 consisting of only the characters a and b, such that the following conditions are satisfied:

1. Both S_1 and S_2 are palindromes.
2. Both S_1 and S_2 should contain exactly X occurrences of a and Y occurrences of b.

If there are multiple possible answers, you may print **any** of them. If it is not possible to construct two distinct strings satisfying the conditions, print -1 instead.

EXPLANATION:

Notice the following:

1. If X is odd, we need to put a right in the middle of S_1 and S_2. Similarly, if Y is odd, we need to put a right in the middle of S_1 and S_2. Therefore, X and Y cannot be odd at the same time.
2. If X is 1, then we can only create one palindrome, therefore X cannot be 1. Similarly, Y cannot be 1.
3. Otherwise, after knowing what the middle character is, build the first half of S_1 by using $\frac{X}{2}$ a characters first, then $\frac{X}{2}$ b characters later; similarly, build the first half of S_2 by using $\frac{Y}{2}$ b characters first, then $\frac{X}{2}$ a characters later.

TIME COMPLEXITY:

Time complexity is $O(X+Y)$ for each test case.

PROBLEM:

Chef has a binary string S of length N. He wonders if it is possible to divide S into exactly K non-empty substrings such that each S_i belongs to exactly one substring and the XOR of each substring is the same. Can you help Chef to determine if it is possible to do so?

Note: XOR of substring $S_{L\cdots R}$ is defined as: $\text{XOR}(S_{L\cdots R}) = S_L \oplus S_{L+1} \oplus \cdots \oplus S_R$.

Here, $\oplus$ denotes the bitwise XOR operation.

EXPLANATION:

The bitwise XOR of any binary string is either 0 or 1. Therefore if an answer exists then it is either K non-overlapping substrings having XOR equal to 1 or K non-overlapping substrings having XOR equal to 0. We can iterate over the binary string and check whether we can divide the string into K non-overlapping substrings having XOR equal to 0 or K non-overlapping substrings having XOR equal to 1
We can greedily select the first $K-1$ substrings having same $XOR(0\ or\ 1)$ and check whether there XOR is equal to the substring formed by the remaining characters. If it is equal output `YES` else `NO`.

We can show that if an answer exists it can always be constructed by this greedy approach.
Suppose there exists an answer such that the first $K-1$ substrings are not selected greedily, let it be $S[1, S_1], S[S_1+1, S_2], \ldots\ldots\ldots\ldots\ldots\ldots S[S_{K-1}+1, N]$, where S_i is the right end of the selected substring. Let there be and index $S_0 < S_1$ such that XOR of substring $S[1, S_1]$ is same as XOR of substring $S[1, S0]$, we select $S[1, S0]$ as the first substring and append the remaining characters of substring $S[1, S_1]$ to the start of the second string. The XOR of $1st$ and $2nd$ substrings remains same as $XOR\ of\ S[1, S_0] = S[1, S_1]$ and $XOR\ of\ S[S_0+1, S_2] = S[1, S_2] \oplus S[1, S_0] = S[S_1+1, S_2]$ as XOR of $S[1, S_1] = S[1, S_0]$ Now move to the next substring and keep on repeating the same procedure until all the $K-1$ substrings are selected greedily $i.e.$ we select the substring as soon as the XOR becomes equal to what we need $(0\ or\ 1)$.

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

Strings, Implementation

PROBLEM:

For a fixed set of Latin letters **S**, for each of given **N** words decide if it can be formed using only letters from **S**.

QUICK EXPLANATION:

For each of given words, check if all its letters are in the given set of known letters **S** by iterating over letters in **S** explicitly or using any data structure with fast lookup operation.

EXPLANATION:

Subtask 1

In the first subtask, the set **S** contains exactly one letter. In this case, it is very easy to decide if a given word can be formed using only letters from **S**, because there is only one letter that can be used, so for example if the letter in **S** is **c**, then all possible words that can be written using it are: **c, cc, ccc, …**. Since the length of any word in the input can be at most 12, then there are exactly 12 different words for which the answer is `"Yes"`. For all other words the answer is **"`No`"**, because each of them contains at least one letter not in **S**.

Subtask 2

In the second subtask, **S** can contain at most 26 letters. In this case, for a given input word **W**, we want to check if all its letters are in **S**. In order to do that, we can iterate over all letters of **W** and for each one check if it is in **S**. Since **S** is very small, that check can be performed actually by iterating over all letters in **S** explicitly. The answer is **"`Yes`"** if all letters of **W** are in **S**, otherwise the answer is **"`No`"**. Thus, for a single word **W**, the answer can be computed in **O(|W| * |S|)** time, so the total running time is **O(|total_len|*|S|)**, where **total_len** is the sum of lengths of all **N** words in the input.

For a faster solution, first we can store letters from **S** in any data structure that provides fast lookup - array will work the best here, since there are at most 26 different letters, but hash table is also fine here. If array is used this step building the array takes **O(|S|)** time. Then, for a given input word **W**, we can check if all of its letters are in **S** using the lookup in our data structure. It follows that the total running time of this method is **O(|S|+|total_len|)**, where **total_len** is the sum of lengths of all **N** words in the input.

PROBLEM:

You are given a binary string A of length N. Rearrange the binary string to form binary strings S and T such that the length of the longest common substring between S and T is **minimum**.

Print both strings S and T.

Note that both S and T must be anagrams of A — see the examples below for more clarity.

EXPLANATION:

Let cnt_0 denotes the number of $0s$ in the string A, and let cnt_1 denotes the number of $1s$. Without loss of generality, let us assume that $cnt_0 \geq cnt_1$.

Consider any string S. The 1s in the string will be partitioning the string into $cnt_1 + 1$ blocks having non-negative number of $0s$. By pigeonhole principle, at least one of the blocks will have $\lceil \frac{cnt_0}{cnt_1+1} \rceil$ $0s$. This means that the longest common substring will atleast have these many 0s. This argument suggests us that we should have our S in such a way that 0s are grouped as evenly as possible.

Examples

So, if $A = 00000011$, we should have $S = 00100100$. If we have $A = 000000011$, we should have $S = 001001000$. For now, let's try to put extra $0s$ towards the end blocks. The reason will soon become clear.

After constructing S, we want to construct T so as to minimize the length of longest common substring. For this, we can greedily group all the $0s$ together, and all the $1s$ together. But should $0s$ come before $1s$ or vice-versa? Note that because of our construction of S, we have greater number of $0s$ towards end. A corner case arises when only the last partition contains larger number of $0s$ as compared to other partitions, as this partition is preceded by a 1 but is not followed by a 1. This suggests that we should first have $0s$, and then $1s$ in T.

Examples

If $A = 000000011$, then we have $S = 001001000$.
Now, if $T = 110000000$, we have a common substring of length 4, as the last 1 of S is also getting matched.
However, if we put $T = 000000011$, we will have a common substring of length 3 only, which is minimum as we discussed above.

TIME COMPLEXITY:

$O(N)$ or $O(N \cdot \log N)$ for each test case, depending on the implementation.

PREREQUISITES:

Count of a unique element in any range of an array using concept similar to prefix sum .

PROBLEM:

You have a binary string S of length N.

You must perform the following operation on the binary string S **exactly once**:

- Choose two integers L and R $(1 \leq L \leq R \leq N)$ and invert the substring $S_{L \ldots R}$ (i.e change 1 to 0 and change 0 to 1).

Determine whether you can make the number of zeroes in S equal to number of ones in S by performing the above operation exactly once. If there exists a way, also output the bounds of the chosen substring.

QUICK EXPLANATION:

Whenever the length of the string is even it is possible to make the count of $zeroes$ and count of $ones$ equal by inverting some prefix of the string.

EXPLANATION:

If N is odd the count of $zeroes$ and $ones$ in the string can never be made equal as addition of two same numbers is even. Therefore when N is odd the answer is always `NO`. Now, we will prove that when N is even it is always possible to invert some prefix of the string to make the count of $ones$ and the count of $zeroes$ in the string equal.
Let P_i represent the prefix of the string S ending at index i $i.e.$ substring formed by the first i characters of the string $S \rightarrow S[1.....i]$. Let the difference between the count of $ones$ and the count of $zeroes$ in string S be a.
Suppose we invert the complete string S the difference now becomes $-a$. Also, Inverting P_1 changes a by 2 and Inverting P_2 changes the difference obtained by inverting P_1 again by 2. Overall, Inverting P_{i+1} changes the difference obtained on inverting P_i by 2. This means by inverting prefixes of different length we can make difference equal to all values from $-a$ to a where adjacent values differ by 2. Let $a > 0$ then some possible differences are $-a, -a+2.....a-2, a$.
Also as the length of the string S is even therefore either both the count of $zeroes$ and count of $ones$ in string S are even or they both are odd. This means that initial difference a is an even number. This means that the possible difference take the value of all even numbers between $-a$ to a which includes 0.
Lets $Ones_i$ represent the numbers of 1 in the prefix of length i. We just need to find an index i such that inverting P_i makes the count of $zeroes$ and the count of $ones$ equal to $N/2$. This boils down to finding an index i such that $Ones_n - N/2 = 2 \cdot Ones_i - i$.

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

You are given two strings S and A of lengths N and M respectively.

String S contains characters from the set {?, a, b, c, d, e}.
String A contains characters from the set {a, b, c, d, e}.
Let S′ denote the string formed by replacing all the ? in S using the characters from the set {a, b, c, d, e}.
Construct S′ such that A is not a subsequence of S′.

If multiple such S′ exist, output any. If no such S′ exists, print −1.

EXPLANATION:

It is obvious that if string A is already a subsequence of string S then there is no way to construct S’ so return -1.

For when it is not a subsequence of S, we keep a track of how much of prefix substring of A has been found as subsequence as we traverse the string. When we encounter a ‘?’ we place a character in S’ which is not same as the next index of the prefix substring. Thus avoiding replacing of a same characters as in string A. Continue this till all ‘?’ have been processed which results in S’ which does not have A as a subsequence.

TIME COMPLEXITY:

$O(|S|)$ for each test case as we traverse the string once.

PROBLEM:

Given a binary string of S of length N. For every occurence of 01 in S, a tax of X rupees will be charged, while for every occurence of 10 in S, a tax of Y rupees will be charged.

Example

For $X = 3, Y = 5$ and $S = 100101$, then S has 2 occurrences of 10 and 2 occurrences of 01, so the tax charged $= 2 \cdot 3 + 2 \cdot 5 = 16$

Rearrange the string S in any way you want. **Minimize** the amount of tax needs to be paid.

EXPLANATION:

Given that for every occurrence of 01 in S, a tax of X rupees is charged, while for every occurrence of 10 in S, a tax of Y rupees is charged. We can rearrange the string in any way we want.

Observation 1
If the given string contains only $0's$ or only $1's$ then there will be 0 occurrences of 10 and 01 in both the cases. Therefore in this case tax charged will always be 0.

Example

- $X = 2, Y = 3, S = 00000$

Tax charged $= 0$.

- $X = 2, Y = 3, S = 11111$

Tax charged $= 0$.

Observation 2
In order to **minimize** the amount of tax needs to be paid the number of occurrences of 10 and 01 in the string should be **minimized**. In order to do that either place all the $1's$ at the start of the string or at the end of the string.

> **CASE 1 :** Placing all the $1's$ at the start of the string ensures that atmost 1 occurrence of 10 and no occurrence of 01 will be present in the string for which the tax charged will be Y rupees.

Example

- $X = 2, Y = 3, S = 1010110$

Initially there are 3 occurrences of 10 and 2 occurrences of 01; therefore, the tax charged will be $2 \cdot 2 + 3 \cdot 3 = 13$.

If we place all the $1's$ at the start of the string, the string S will become 1111000. Now the number of occurrences of 10 will become 1 and number of occurrences of 01 will become 0. Therefore, the tax charged will be $0 \cdot 2 + 1 \cdot 3 = 3$.

CASE 2 : Similarly, placing all the $1's$ at the end of the string ensures that atmost 1 occurence of 01 will be present in the string for which the tax charged will be X rupees.

Example

- $X = 2, Y = 3, S = 1010110$

Initially there are 3 occurrences of 10 and 2 occurrences of 01; therefore, the tax charged will be $2 \cdot 2 + 3 \cdot 3 = 13$.

If we place all the $1's$ at the end of the string, the string S will become 0001111. Now the number of occurrences of 10 will become 0 and number of occurrences of 01 will become 1. Therefore, the tax charged will be $1 \cdot 2 + 0 \cdot 3 = 2$.

Minimum tax charged using **case 1** is Y rupees and minimum tax charged using **case 2** is X rupees. Since we can rearrange the string in any way we want, we will rearrange it according to the case which gives minimum tax charged. Therefore final output will be minimum of X and Y if string contains both $0's$ and $1's$.

TIME COMPLEXITY:

$O(1)$ for each test case.

PROBLEM:

A binary string is called alternating if no two adjacent characters of the string are equal. Formally, a binary string T of length M is called alternating if $Ti \neq Ti+1$ for each $1 \leq i < M$.

For example, 0, 1, 01, 10, 101, 010, 1010 are alternating strings while 11, 001, 1110 are not.

You are given a binary string S of length N. You would like to rearrange the characters of S such that the length of the longest alternating substring of S is maximum. Find this maximum value.

A binary string is a string that consists of characters 0 and 1. A string a is a substring of a string b if a can be obtained from b by deletion of several (possibly, zero or all) characters from the beginning and several (possibly, zero or all) characters from the end.

EXPLANATION:

To form the given string of maximum length, we calculate the number of 1's and 0's in the original string.
Let n_1, n_0 be the counts respectively.

Case1: $n_0 = n_1$
(Keeping either '0' or '1' in the beginning and alternating)
$\implies ans = 2 * n_1$

Case2: $n_0 < n_1$
(Keeping '1' in the beginning and alternating)
$\implies ans = 2 * n_0 + 1$

Case3: $n_0 > n_1$
(Keeping '0' in the beginning and alternating)
$\implies ans = 2 * n_1 + 1$

The above three cases can be mathematically combined in a single equation as:
$ans = 2 * min(n_0, n_1) + (n_1 ! = n_0)$

TIME COMPLEXITY:

The above calculation can be done in constant time. However, calculating n_1 and n_0 requires a complete iteration of the string. Hence, the solution has a time complexity of $O(N)$.

PROBLEM:

Chef is working on his swap-based sorting algorithm for strings.

Given a string S of length N, he wants to know whether he can sort the string using his algorithm.

According to the algorithm, one can perform the following operation on string S any number of times:

- Choose some index i $(1 \le i \le N)$ and swap the ith character from the front and the ith character from the back.
  More formally, choose an index i and swap S_i and $S_{(N+1-i)}$.

For example, $\underline{\text{d}}\text{cb}\underline{\text{a}}$ can be converted to $\underline{\text{a}}\text{cb}\underline{\text{d}}$ using one operation where $i = 1$.

Help Chef find if it is possible to sort the string using **any** (possibly zero) number of operations.

EXPLANATION:

Let us first find out the string that the Chef will end up with, given that he performs the swap optimally.

Let us consider the index i and $j = (N + 1 - i)$. Also, let $i \le j$. If $S_i \le S_j$ then we do not need to swap these characters (as they are in already in correct order). Otherwise, we will swap S_i and S_j.

Let us first perform the above operation by iterating over the complete string, and let us denote the final string as $S^{'}$. If $S^{'}$ is sorted, then our answer is **YES**, otherwise our answer will be **NO**.

To check if $S^{'}$ is sorted, we can iterate over the string and check that $S_i \le S_{i+1}$ for all i such that $1 \le i \le N - 1$. Another way to check is to use the $sort$ function which is usually present in the Standard Template Library (or similar libraries) in most of the languages.

TIME COMPLEXITY:

$O(N)$ or $O(N \cdot \log N)$ for each test case, depending on the implementation details.

PREREQUISITES:

Inversions

PROBLEM:

Chef is given a binary string A of length N. He can perform the following operation on A any number of times:

- Choose L and R $(1 \le L \le R \le N)$, such that, in the substring $A[L, R]$, the number of 1s is **equal** to the number of 0s and **reverse** the substring $A[L, R]$.

Find the lexicographically **smallest** string that Chef can obtain after performing the above operation **any** (possibly zero) number of times on A.

String X is lexicographically smaller than string Y, if either of the following satisfies:

- X is a prefix of Y and $X \neq Y$.
- There exists an index i such that $X_i < Y_i$ and $X_j = Y_j, \forall j$ such that $1 \le j < i$.

EXPLANATION:

The first intuition when we read the problem is, can we sort the string? If Yes, we have got our answer, as the sorted string is the lexicographically smallest possible string that we can get.

Let's see if we can sort the string. Let there be an index i such that $S_i = 1$ and $S_{i+1} = 0$. If there is no such index i, then we have already sorted our string. Otherwise, the substring S_iS_{i+1} is a valid substring, and we can reverse it, making it 01. We can repeat this process as long as we can find such index i, and hence finally ending up with a sorted string.

But how to prove that the above process will eventually terminate?

We will prove this by using the number of inversions. It is recommended to first understand the concept of inversions.

Note that when we perform the above operation, the number of inversions decreases by 1. Because the initial number of inversions is finite, and the final number of inversions will be greater than or equal to 0, we can perform the above operation only finite number of times, eventually reaching a state where we cannot find a valid index i. In other words, the number of inversions will become 0, and hence we will end up getting our sorted string.

TIME COMPLEXITY:

We need to sort the given string - $O(N \cdot \log N)$ for each test case.

PROBLEM:

Chef has two binary strings A and B, each of length N. He can perform the following operation on A any number of times:

- Choose L and R $(1 \le L \le R \le N)$, such that, in the substring $A[L, R]$, the number of $1s$ is **not equal** to the number of $0s$ and **reverse** the substring $A[L, R]$.

Find whether Chef can convert the string A into the string B by performing the above operation **any** (possibly zero) number of times on A.

QUICK EXPLANATION:

- If A and B are not anagrams, then the answer is **NO**
- If any one of A or B has only alternate $0's$ and $1's$ (like *101010* or *010101*), then the answer is **YES** iff $A = B$, otherwise answer is **NO**
- Otherwise, there exists an index i such that $A_i = A_{i+1}$. Similarly, there exists an index j such that $B_j = B_{j+1}$.
- We can show that if there is an index i such that $A_i = A_{i+1}$, we can sort the string A using the above operations. Similarly, we can sort B, and because the operations are reversible, we can show that A can be converted to B (by first sorting A, and then following the operations used in sorting B in reverse order).

EXPLANATION:

Because the mentioned operation cannot change the number of $0's$ or $1's$ in the string, the answer will always be **NO** if A and B are not anagrams.

Now let's say that for every $i(1 \le i < N), A_i \neq A_{i+1}$. In other words, A has alternate $0's$ and $1's$. In this case, note that every even length substring has equal number of $0's$ and $1's$, and therefore we can only reverse substrings of odd lengths. However, every odd length substring is a palindrome, and hence reversing it doesn't modify A. In simpler words, we can never modify A using the above operations. So, the answer will be **YES** only if $A = B$, otherwise the answer will be **NO**.
Note that the operations are reversible, and converting A to B is equivalent to converting B to A. Hence, if B has alternate $0's$ and $1's$, then the above argument holds.

Now we have an index i such that $A_i = A_{i+1}$. Similarly, we have an index j such that $B_j = B_{j+1}$. We will show that if there is such an index, then we can sort our string. If this is true, we can first sort A. Let S be the ordered set of operations which are applied to B while sorting it. Because of the reversible nature of the operation, we can apply the operations of S in the reverse order to string A, and get the string B. Note that this is true because A and B are anagrams.

Sorting string A when $A_i = A_{i+1}$
Without loss of generality, let $A_i = 0$

Bringing all 1's which are present on the left side of the pair to the right side
Bringing all 0's which are present on the right side of the pair to the left side

If we have $A_i = 1$, then we essentially need to do the same thing. Hence we have shown that if there are two consecutive equal characters, we can sort our string, and therefore, we can convert A to B as explained above.

TIME COMPLEXITY:

$O(N)$ or $O(N \cdot \log N)$ for each test case, depending on the implementation.

PROBLEM:

You are given two strings **S1**, **S2** with equal lengths. Both strings, on any position, can contain a lowercase latin letter or a question mark '?'.

A question mark can be equal to any of lowercase latin letters. We define the difference between the strings as the number of positions ***i***, such that the strings are different on this position. Your task is to compute the minimal and the maximal difference between the strings.

QUICK EXPLANATION:

Let's consider any position ***i*** in the strings. Since a question mark can be any letter, if at least one of the strings have a question mark on the i-th position, we can always change it to match the letter in the other string or we can always change it to a letter which produces a mismatch. Therefore, the minimal difference is the number of positions for which none of the strings contain a question mark and there is a mismatch between them on this position. On the other hand, the maximal difference is the length of the strings reduced by the number of positions for which none of the strings contain a question mark and there is a match between them on this position.

EXPLANATION:

We define a **strong match** as the position i, such that none of the strings contain a question mark at the i-th position, and the letters in the strings at the i-th position match.

We define a **strong mismatch** as the position i, such that none of the strings contain a question mark at the i-th position, and the letters in the strings at the i-th position do not match.

Let's consider any position i in both strings. If at least one string contain a question mark on the i-th position, we can create a match or a mismatch between the strings at this position.

There are basically two cases to consider:

1. If only one string contains a question mark on the i-th position.

   Without loss of the generality, let's assume that S1[i] = '?' and S2[i] = 'a'. Since the latin alphabet has 26 letters, we can produce a mismatch by setting up S1[i] to any letter different by 'a' and we can produce a match by setting up S1[i] to 'a'.

2. Both strings contain a question mark on the i-th position

   Since the latin alphabet has 26 letters, we can produce a match by setting up S1[i] = S2[i] = 'a' and we can produce a mismatch by setting up S1[i] = 'a' and S2[i] = 'b'.

Based on these two facts, we for any position which is not a strong match or a strong mismatch, we can make it a match or a mismatch.

Therefore, the minimal difference is the number of strong mismatches. On the other hand, the maximal difference is the length of the strings reduced by the number of strong matches.

Time complexity:

Linear in terms of the length of input strings.

PROBLEM:

Given two strings A, B and Q queries, for each query we have to choose substring $A[L...R]$ and see if it is possible it to convert to $B[L..R]$ by cyclically incrementing two consecutive characters of A[L…R].

EXPLANATION:

Each query can be solved in O(R-L) as:
For query: L, R.
We will iterate over the substring from lowest to highest index and use the operation on $[i, i+1]$ to change $A[i] = B[i]$ for $i < R$.
If the last character $A[R]$ becomes equal to $B[R]$ after doing above operations the answer is YES else it is NO.

The above process of updating at each index and be mathematically expressed as:
Let $C_i = B_i - A_i$, be the number of operations to convert A_i to B_i if increasing single character were allowed.
Also for each query let's define D_i as the number of operation we apply at index i.
Then, mathematically $D_i = C_i - D_{i-1}$ as after applying D_{i-1} operation on index $i-1$, A_i changes to $A_i + D_{i-1}$, so to make it equal to B_i we do $B_i - (A_i + D_{i-1})$ operations on i^{th} index.

So, we get the value of D_R as:
$D_R = C_R - C_{R-1} + C_{R-2} + ...(-1)^{R-L} C_L$
As we cannot apply any operation at R^{th} position so if we want substrings to be equal D_R must be 0. Therefore the alternating sum of C_i in range L to R must be zero modulo 26 for answer to be YES .
Modulo 26 is required because operations are cyclic with length 26, as applying the same operation to an index 26 times, results in the same result. So, if $D_R = 26 \cdot p + t$ (where t is the remainder of D_R modulo 26), then applying operation at R, D_R times is same as applying operation at R, t times.

PREREQUISITES:

Familiarity with comparing string lexicographically will be useful.

PROBLEM:

You are given a string S of length N, containing lowercase Latin letters. You are also given an integer K.

You would like to create a new string $S^{'}$ by following the following process:

- First, partition S into exactly K non-empty subsequences $S_1, S_2, \ldots, S_K$. Note that every character of S must be present in exactly one of the S_i.
- Then, create $S^{'}$ as the concatenation of these subsequences, i.e, $S^{'} = S_1 + S_2 + \ldots + S_K$, where $+$ denotes string concatenation.

Determine the **lexicographically** smallest $S^{'}$ that can be created.

EXPLANATION:

We can divide the solution in two parts, when $K = 2$ and when $K > 2$.

Creating lexicographically smallest string when $K > 2$

Let α represents the lexicographically smallest character, and β represents the second lexicographically smallest character in the string.

We want to choose a subsequence such that the resulting string is lexicographically smallest. To do this, we will greedily choose all the occurrences of α in our first subsequence. Let the last occurrence of this character be ind.

What to choose from the remaining string after choosing the all the lexicographically smallest characters?

We will start choosing all the occurrences of β that occurs after the index ind. If β also occurs before the index ind, then we are done with this subsequence. Otherwise, we have selected all the occurrences of β, and continue with the third lexicographically smallest character.

Once we have chosen all the characters for the first subsequence, we can remove these characters from the string, decrease K by 1, and continue our problem.

Creating lexicographically smallest string when $K = 2$

We want to choose a subsequence such that the resulting string is lexicographically smallest. To do this, we will first consider the lexicographically smallest character in the string. We will greedily choose all these characters in our first subsequence.

What to choose from the remaining string after choosing the all the lexicographically smallest characters?

Suppose till now, we have chosen the complete prefix of our string as part of first sub-sequence. In that case, we have a remaining smaller string and we can solve it by starting the logic from the beginning.

Otherwise, we have atleast one character that we have not taken in the sub-sequence. The first such character will be at the beginning of the second subsequence. Let us represent this character by α. Now, we consider one of the character β of the remaining string.
If that character is lexicographically greater than α, we will not take that character in the first subsequence greedily, so that we are not placing it before α in the final string.

Now, consider the case when β is lexicographically smaller than α. If β is the lexicographically smallest element in the remaining string, we will take β in the first subsequence. Otherwise, we will skip this character in order to take the lexicographically smaller character in the sub-sequence.

Note that the condition subsequences should be non-empty is redundant. If we can find a solution when empty sub-sequences are also allowed, we can convert that solution into non-empty sub-sequences by breaking down our previous sub-sequences.

TIME COMPLEXITY:

We need at most 26 non-empty subsequences to get our answer. So we can implement the above approach in $O(N \cdot 26)$ for each test case.

PROBLEM:

You are given a binary string S of length N.

You have to perform the following operation **exactly** once:

- Select an index i $(1 \le i \le N)$ and delete S_i from S. The remaining parts of S are concatenated in the same order.

How many **distinct** binary strings of length $N-1$ can you get after applying this operation?

QUICK EXPLANATION:

- Let str_i represents the string obtained after deleting the ith character from the string. Compare str_i and str_{i+1}. When are they same? When are they different? What if we consider str_i and str_j?

EXPLANATION:

Let us use the notations as defined in the Quick Explanation and try to answer the questions that are asked there.

Consider a continuous block of $0s$. If we remove any 0 from this continuous block, then the resulting strings will be equal. The same holds for a continuous block of $1's$. So, if $S_i = S_{i+1}$, then the resulting str_i will be equal to str_{i+1}.

This motivates us to consider the string as the blocks of $0s$ and $1s$. Let us represent the string as $S = O_1Z_1O_2Z_2...O_KZ_K$, where Z_i represents a block of $0s$ and O_i represents a block of $1s$. (It is possible that string starts with 0 or ends with 1, all these cases are equivalent, so the above representation is good enough for us).

We have seen that if we remove any 1 from O_i, the resulting strings will be equal. The only case to consider is, if we remove 1 from O_i in one case and from O_j in second case $(i \neq j)$.
The first string will be: $O_1Z_1...Z_{i-1}(O_i - 1)Z_i...O_jZ_j...O_KZ_K$
The second string will be: $O_1Z_1...Z_{i-1}O_iZ_i...(O_j - 1)Z_j...O_KZ_K$

We can see that strings differ at the O_i block. So these both strings are different.
Therefore, total number of **distinct** strings that we can obtain is equal to the number of continuous blocks of $0s$ and $1s$.

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

A string A of length N is said to be **anti-palindrome**, if for each $1 \leq i \leq N$, $A_i \neq A_{(N+1-i)}$.

You are given a string A of length N (consisting of lowercase Latin letters only). Rearrange the string to convert it into an anti-palindrome or determine that there is no rearrangement which is an anti-palindrome.

If there are multiple rearrangements of the string which are anti-palindromic, print **any** of them.

QUICK EXPLANATION

What is the condition for NO answer?

If the string is of odd length the answer must be NO, because of the centremost letter. Otherwise, intuitively we can observe that if a letter occurs more than n/2 times there can not be any rearrangement possible. Can be also proved by pigeonhole priciple.

How to rearrange if no letter appears > n/2 times in an even length string?

We can sort the string. Now we reverse the right half of the string, that way letters at a[i] and a[i +n/2] are against each other. Since no letter appears > n/2 times both these characters are different.

EXPLANATION:

For a string with odd length the answer is always NO because of the centremost character. So, from now on we'll deal with the even length strings.

If any character appears more than n/2 times, using pigeonhole principle we can conclude that no arrangement is possible. If it is not the case, we can make a arrangement by sorting the string and putting the a[i] and a[i + n/2] characters against each other. So, our rearrranged string will look like a[1], a[2], … a[n/2], a[n], a[n-1] … a[n/2 + 1].

TIME COMPLEXITY:

O(n) for each test case.

PREREQUISITES:

Binary Lifting

PROBLEM:

Consider the following string of infinite length: `abc...xyzabc...xyz...`

Let string B be the prefix of this string of length M. For e.g. if M=6, B= `abcdef`.

Vedansh calls a string X of length K special if it satisfies the following conditions:

- B is a substring of X. Formally, for some $1 \leq L \leq R \leq K, X_{L...R} = B$

Vedansh has string A of length N (consisting of lowercase Latin letters only) and he wants to convert it into a special string. To do so he can perform the following operation:

- Pick an i such that $1 \leq i \leq |A|$ and delete A_i. The remaining parts of A are concatenated.

Since Vedansh is lazy, he wants to do this in minimum number of operations or determine if it can not be converted to a special string. Help Vedansh in doing so.

QUICK EXPLANATION:

What if we fix the position of the first character of B in X?

We will pick the remaining characters greedily by choosing the next character at its next occurrence. And thereby, the subsequence corresponding to B gets fixed. We can just delete the unwanted characters between the first and last character of B.

How to check optimally for all starting positions?

If we keep picking the next occurrence in X of the next character of B, we'll have to make O(n) computations each time. To do this optimally we'll have to store some information prior to this step.
We can store the position of later characters in multiple jumps.

Which technique and data structure to use to perform these operations optimally?

The binary lifting technique and the corresponding dp values provide this functionality to check positions of characters in jumps of 1,2,4 and so on. We can use this to find the position of character after M-1 positions and then pick the result that uses the minimum number of delete operations.

EXPLANATION:

The string B will appear in string X as a substring if there is a subsequence B in the string X. So corresponding to a particular subsequence B, we will have the delete the unwanted characters in between the first and last character to obtain B as a substring. We have to pick such a subsequence in which the difference between the position of the first and last character is minimum, thereby leading to the minimum deletions from X.

If we fix the position of the first character of B in X(in this case 'a'), the next character should be chosen as near as possible to minimize the deleted characters. If we proceed in the above manner the whole subsequence gets fixed and hence the position of characters of B.
So we want to fix the first character at every position possible and check the difference b/w the first and last character of the resultant subsequence and pick the one with the least difference, hence the minimum deletions. If we carry out these operations individually and directly we would require O(n) time at each step giving time limit errors.

To carry out these operations optimally we have to make some calculations and store some info prior to this step. In the naive approach, we calculate the position of the next element every time and repeat this step. By invoking our knowledge of binary lifting, we try to calculate the position of the next to next element, the 4th next element, and so on. This can be easily calculated using the dynamic programming technique.

After storing these values we can calculate the (m-1)th next element from each starting point using the binary lifting technique and pick the subsequence that requires the minimum deletions.

TIME COMPLEXITY:

O(nlogn)

PROBLEM:

Check if a given string s can be changed to another given string t, given that the only mutations possible in s are that vowels can be replaced by other vowels and consonants can be replaced by other consonants.

QUICK EXPLANATION:

If the strings s and c are of same length and have vowels and consonants at the same indices, then they are mutable, otherwise not.

EXPLANATION:

Since we cannot add or remove any characters from the string s to mutate it to string t, hence if the lengths of the strings s and t are unequal, we can simply print **no**. No matter what characters we replace in the string s it will never be equal to t as the number of characters are different.

We only need to check further if the length of both the strings is same. In that case we need to check if both the strings have vowels and consonants at the same places. For example: “abc” and “efg” have vowels at first index and consonants on the remaining ones. Therefore we can change a to e (vowel to vowel), b to f (consonant to consonant) and c to g (consonant to consonant) and make the string “abc” equal to “efg”. But if the vowels and consonants in one string are at different positions than the other one, then it will be impossible to change the first string to be equal to the second since we can only replace vowels with vowels and consonants with consonants. For example if string s is “abc” and string t is “def”, then we can’t do anything to change string s to be equal to t. ‘a’ can only be replaced by a vowel, but in the string t, the first letter is a consonant, i.e., ‘d’. Hence it is impossible to change string s to string t.

This gives us the logic that we need to implement:

```
if length of s ≠ length of t, print no.
else
    set a flag variable to true
    loop through the strings
        if s[i] is a vowel and t[i] is a vowel or if s[i] is a consonant and t[i] is a consonant then continue.
        else set flag to false, print no and exit the loop.
    if flag is set to true at this point, print yes
end
```

PREREQUISITES:

String Manipulation

PROBLEM:

$Mr.Hannibal$ is a Software Engineer, with specialization in Artificial Intelligence. He is really annoyed by the words that have a mix of **Upper and Lower case characters**.

Thus he wanted to build a software that would change the letters' register in every word so that it either **only** consists of lowercase letters or, vice versa, only of uppercase ones.

Mr.Hannibal also wants to make sure that $the\ least\ number$ of characters are **updated**. For example, the word **ColOr** must be replaced with **color**; and the word **aPPlE —** with **APPLE**. In case a word contains $equal\ number$ of uppercase and lowercase characters, he should replace all the letters with $lowercase$ ones. For example, **bOoK** should be replaced by **book**.

You being the $assistant\ to\ Mr.Hannibal$, help him develop the piece of software.

EXPLANATION:

Given a string, we only need to count the number of lower and upper case characters. The one with higher number of lower case characters must be fully converted to lower case string ,and vice versa.

TIME COMPLEXITY

The time complexity is $O(N)$

PROBLEM:

You have acquired a dictionary of N words of a forgotten language. Meanwhile, you also know K phrases used in modern languages. For each of the words of the forgotten language, your task is to determine whether the word is still in use in any of these K modern phrases or not. Each phrase has L words given in input.

EXPLANATION:

==========
First, you just need to store all the N words of the forgotten language and all the phrases. For each each word, then, you can traverse over all phrases and check if its present in any one of them. You can also build a single array of all the words from all phrases and then check for each forgotten language word if its present in the array or not.

If you want it to more efficient you can store all phrase words in a set where insertion is $O(\log(\text{size}))$ and checking if a string is present or not is also $O(\log(\text{size}))$.

For implementation, you need to know basic programming concepts and syntax of a language of your choice. You should know about arrays, conditionals, loops, input/output to solve this problem.

First, we include implementations in three popular programming contest languages.

C++ code

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    int T;
    cin >> T;
    for (int cas = 1; cas <= T; cas++) {
    // we use 'cas' because 'case' is a keyword
    int N, K, L;

    // allocate more than necessary
    // note that phrases[i] is a vector of strings.
    vector < string > phrase[55];

    //array of forgotten words
    string forgotten[109];

    cin >> N >> K;
    for (int i = 0; i < N; i++) {
        cin >> forgotten[i];
    }

    for (int i = 0; i < K; i++) {
        cin >> L;
        for (int j = 0; j < L; j++) {
            string S;
            cin >> S;
            phrase[i].push_back(S);
        }
    }
```

Java Code

Note that java.util.Scanner provides an easy way to tokenize and read the input. However, for problems with huge inputs and strict time limits, it is not recommended because it is slow. Instead, one should use BufferedReader, like so:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.*;

public class Main {
    static String[] data;

    public static void main(String[] args) throws IOException {
    BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(System.in));
    int testCases = Integer.parseInt(bufferedReader.readLine());

    while (testCases-- > 0) {
        //get N and K
        int forgottenLanguageWordsCount, modernLanguagePhrasesCount;
        data = bufferedReader.readLine().split(" ");
        forgottenLanguageWordsCount = Integer.parseInt(data[0]);
        modernLanguagePhrasesCount = Integer.parseInt(data[1]);

        //build array
        String[] forgottenWords = bufferedReader.readLine().split(" ");

        //build a single list with all words from all phrases
        List<String> modernWordsList = new ArrayList<>();
        for (int i = 0; i < modernLanguagePhrasesCount; i++) {
            data = bufferedReader.readLine().split(" ");
            int totalWordsInPhrase = Integer.parseInt(data[0]);

            //add to list
```

Python code

```
T = input()
for cas in xrange(1,T+1):
    N, K = map(int, raw_input().strip().split())

    forgotten = raw_input().split()

    allWordsfromPhrases = []
    for i in xrange(K):
    allWordsfromPhrases += raw_input().split()

    ans = ""
    for i in xrange(N):
    if forgotten[i] in allWordsfromPhrases: ans += "YES "
    else: ans += "NO "

    print ans
```

Python is ridiculously easy in this problem. Note that the python code has a bit of different flavor, specifically the for…else statement.

Suggestions

I will directly quote kevinsogo 17 from one of his editorials.

Now that you've learned that many, many things can go wrong even for such a simple problem, how does one go about preventing it?

Well, for one, it is usually hard to write a completely correct code in the first try. Therefore one should always test the code before submitting it! For example, use the sample data. The sample data is provided for two reasons:

- To ensure that you are following the input/output format correctly, and
- To ensure that you at least get some small inputs correctly.

However, if you pass the sample input, it doesn't mean you'll pass the actual input! So it still lies upon the contestant to ensure that their program will pass whatever kind of test case the judge can give to it. So if the judge returns WA, try making some inputs yourself, until you find one where your program fails. Then start debugging from there. In other problems, it is sometimes useful to make random input generators.

Also, there is the occasional problem that tests how well you follow instructions and handle many tricky edge cases. In such problems, speed/minute optimizations are secondary to correctness, so things like readability/modularity 4 more important, at the expense of efficiency. See the sample programs above as examples, which were written partly with readability in mind.

COMPLEXITY

==========

If we implement a brute force solution, there are worst case $\text{MAXL} * \text{MAXK}$ words and for each forgotten word, we do a linear search. So total complexity is $\text{MAXL} * \text{MAXK} * \text{MAXN}$.
Using set, complexity can be reduced to $\text{MAXL} * \text{MAXK} + \text{MAXN} * \log(\text{MAXL} * \text{MAXK})$.

PROBLEM:

Because of corona there is lockdown outside in chefs city. chef is willing to go outside of home for some work . but because of lockdown chef will go out if there is safe atmosphere outside ,otherwise he will not go outside.

The condition for safe atmosphere is number of corona negative patients should be greater than or equal to number of corona positive patients.

we have given binary number S in which 1 denotes positive patients and 0 denotes negative patients. [you can use first line of each test case as a string]

QUICK EXPLANATION:

Check each number of string, whether it is ‘1’ or ‘0’. if count of 0 is greater than or equal to 1 print YES , NO otherwise

EXPLANATION:

by applying for loop, from 0 to s.size(). check s[i] == 0 then increment in cnt1, else increment in cnt2 if cnt1>= cnt2 print YES otherwise NO

PROBLEM:

Find the number of characters which occur in both given strings.

QUICK EXPLANATION:

For each string find the multiset of characters present in the string. The answer is the number of common elements in the two multisets. Since there are only 62 (26 uppercase alphabets + 26 lowercase alphabets + 10 digits) possible characters, the multiset of characters present in a string can be represented by an integer array of size 62.

EXPLANATION:

Since we must find the number of common characters present in both strings, the order can be any. For each string one needs to find the multiset of characters which appear in it. The intersection of the two multisets will be the longest pattern. Remember, in a multiset the same character may appear more than once.

PREREQUISITES:

String manupulation

PROBLEM:

It's a New Year,and Meena has established some goals for the upcoming year.

He decided to build his own YouTube channel and posted some of his films there. Meena now wants to keep track of all of his YouTube videos so he decided to save all the links to his videos, but youtube links have a lot of unnecessary strings so he decided to store only the important part of the link that is link id.

You are given String SS.

Help Meena to separate link id from a given string.

```
Note! You will be given link in two formats
```

- Format1="Taarak Mehta Ka Ooltah Chashmah - Episode 624 - Full Episode - YouTube"

and

- Format2="https://www.youtube.com/user/dashboard/view/HScledQrT"

here (SCeliffla_k) and (HScledQrT) are Id's respectively.

QUICK EXPLANATION:

In this Question, we have to check whether the string contains the keyword "view" or "watch" and according to that we have to extract the string Id from the link

EXPLANATION:

In this Question, we have to check whether the string contains the keyword "view" or "watch" .
If the string contains "view" the print Id present after the last "/" or if the string contains "watch" then print id present after "=" and before last "/".

PROBLEM:

Given a string of alphanumeric characters, print the sum of the digits.

EXPLANATION:

This is the easiest problem in the contest. Straightforward solutions will likely pass.

The idea is to iterate through the input string, filter out the non-digits, convert the digit characters into actual integers, and sum them. The following are a few sample implementations.

C:

```
#include <stdio.h>

char s[1111];
int main() {
    int cases, cas, i, total;
    scanf("%d", &cases);
    for (cas = 0; cas < cases; cas++) {
        scanf("%s", s);
        total = 0;
        for (i = 0; s[i]; i++) {
            if ('0' <= s[i] && s[i] <= '9') total += s[i] - '0';
        }
        printf("%d\n", total);
    }
    return 0;
}
```

C++:

```
#include <iostream>
using namespace std;

int main() {
    int cases;
    cin >> cases;
    for (int cas = 0; cas < cases; cas++) {
        string s;
        cin >> s;
        int total = 0;
        for (int i = 0; s[i]; i++) {
            if ('0' <= s[i] && s[i] <= '9') total += s[i] - '0';
        }
        cout << total << endl;
    }
    return 0;
}
```

Java:

```
import java.util.Scanner;
public class Main {
    public static void main(String args[]) {
```

```
        Scanner sc = new Scanner(System.in);
        int cases = sc.nextInt();
        for (int cas = 0; cas < cases; cas++) {
            String s = sc.next();
            int total = 0;
            for (int i = 0; i < s.length(); i++) {
                char ch = s.charAt(i);
                if ('0' <= ch && ch <= '9') {
                    total += ch - '0';
                }
            }
            System.out.println(total);
        }
    }
}
```

Implementation details

How to check if a character is a digit

All the above codes use the snippet `'0' <= ch && ch <= '9'` to determine if a character `ch` is a digit. This works because the `char` type in C, C++ and Java are implemented as a single byte representing the ASCII code of the character, and the digit characters `'0'` to `'9'` are consecutive ASCII codes. In fact, you can use a similar technique to check if `ch` is a lowercase character: `'a' <= ch && ch <= 'z'`.

How to convert a character to an integer (if it is a digit)

Since characters are represented simply as bytes, we can perform arithmetic with them. This gives us a way to convert a digit character to a number. To convert a digit character `ch` to a number from 0 to 9, we simply use `ch - '0'`.

Note that a character being represented as a byte is not universal. In some other languages such as Python or Ruby, characters are actually represented as strings of length one, and although comparison between chars still work, you can't necessarily perform arithmetic between chars. For example, in Python, `'l' - 'a'` would be an error. But usually there are still ways to get the ASCII value of a character. For example, in Python, you would use the `ord` and `chr` functions to convert to and from an integer, respectively.

Common errors

This is the easiest problem in the contest. However, that doesn't mean nothing can go wrong with your submission. Lots of things can go wrong, from major problems down to simple gaffes such as choosing the wrong language while submitting! You must be prepared to handle these mistakes, because debugging is part of a programmer's life, and in fact usually takes more time than actual coding.

One common error is forgetting to initialize variables, as in the following:

```
#include <stdio.h>

char s[1111];
int main() {
    int cases, cas, i, total = 0;
    scanf("%d", &cases);
    for (cas = 0; cas < cases; cas++) {
        scanf("%s", s);
        for (i = 0; s[i]; i++) {
            if ('0' <= s[i] && s[i] <= '9') total += s[i] - '0';
```

```
        }
        printf("%d\n", total);
    }
    return 0;
}
```

This one could be quite misleading because the line `total = 0` initializing the `total` variable is present. It's just that it's outside the loop, so it doesn't reset to `0` for the following test case! Consequently, your program will most likely get wrong answers for all but the first test case.

One way to detect this is possibly to add an identical test case in your sample input. If the answers you got are different, then most likely something's wrong. Also, to be sure, declare variables only on scopes they're needed, as shown in the C++ and Java examples above.

Another error is allocating an array too small for the input string. Remember that when given constraints such as $|S| \leq 1000$, judges will most likely test out extremes, so allocating an array of size 200 will not be enough. In fact, for C or C++ an array of size 1000 might not be enough too, because strings in those languages are terminated with a null character `\0`. You need at least 1001 for the null character to be accommodated. (In fact, the examples above use 1111 just to be extra cautious.)

Readability

Another way to help you debug your code is to ensure your code is readable and clear. This includes formatting your code properly and possibly adding a few comments. In general, try to put yourself in your teammate's shoe and see if he/she will understand the code just by reading it, without you explaining it.

Here are some things about readability related to this problem:

- When hardcoding the digits, don't write the ASCII values. Instead, use the actual character `'0'`, `'a'`, etc. Technically, `'0'` would work the same in most cases as `48` (`'0'`'s ASCII code), but `'0'` is much clearer than a random `48` found in code.
- Simplicity is important. There's no need for very advanced programming patterns for something as simple as this. If you check the examples above, you find that they're simple and straight to the point, and it also has the desirable consequence of being clear and readable.
- Some submissions convert chars to ints by first converting the char to a string and then parsing it into an int. Please don't do that. It's harder to read than the methods above, and is actually much more inefficient.
- If possible, try to use library functions that will help you with the task if you know them. In actual real life coding work, it always pays to check whether a particular task already has a builtin function for it or has already been implemented in a library. In our case, you could use `Character.isDigit` in Java or `isdigit` in C++ to check whether a char is a digit.

Miscellanea

Here we mention other common errors and things to remember.

- Don't forget to print a newline at the end of each test case! To be sure, try it yourself first with a couple of test cases and see if they output as expected. Command line tools such as `diff` or `fc` are helpful for this task and guards against manual checking which is unreliable.
- Use zero-indexing. It's idiomatic in many languages including C, C++ and Java. Even though some problems use one-indexing, it's usually helpful to convert to zero indexing in most languages, especially if you're gonna use some builtin functions that assume zero indexing.
- Don't print unnecessary stuff like "Please enter the number of test cases: ". These are considered part of your output and so your solution will be marked wrong. The goal is to *exactly match* the contents of the judge's output file.

Language-specific things

- When using `scanf` to take a string, use `scanf("%s",s);` instead of the incorrect `scanf("%s",&s);`.
- When copy-pasting Java code to be submitted to CodeChef, use `Main` as your public class name (as stated in Sample Solutions | CodeChef). It's because Java requires the file name and public class name to be the same, but since you copy-pasted code, CodeChef doesn't know the public class name and could have a harder time figuring it out from code. It's easier to just assume it's something, say, `Main`.

Contest-specific stuff

These are things usually good to be done during contests but are worth keeping in mind that they are considered bad practice in industry code.

- In Java, there are times when some methods have checked exceptions, such as `IOException` for `BufferedReader.readLine()`, requiring you to handle them with bulky `try-catch` blocks. In these cases, you can just declare `throws Exception` in your main method. This is very much not recommended in production code, but in a contest, it's very convenient.
- In Java, instead of importing each class separately, you can import everything from a single package using "import *", e.g. `import java.util.*;`. Of course this is not recommended at work because it makes it harder to keep track of where each used class is defined, but during contests you don't really need to worry about it.
- Instead of having to type `std::cin` all the time, you can add `using namespace std` in your C++ code so you can just use `cin`. It's also considered bad practice [2] in real life. Nonetheless, it's also convenient, makes code a bit clearer, and can cut down typing time. Even then, people using it must be wary, because problems such as this and this might occur.

Time Complexity:

$O(N)$

PREREQUISITES:

Sliding Window, String, Math

PROBLEM:

Chef is hungry, so he went to a market that has a single row of N Cakeshops arranged from left to right. Every Cakeshop has some cakes.

Chef wants to buy as many cakes as possible. However, the Cake Shop owners has some strict rules that Chef must follow:

- You only have two bags and each bag can only hold a single type of cake. There is no limit on the amount of cake each bag can hold.
- Starting from any Shop of your choice, you must buy exactly one cake from every shop (including the start shop) while moving to the right. The bought cake must fit in one of your bags.
- Once you reach a shop with cake that cannot fit in your bag, you must stop.

Given the type of cake shops have, help Chef find the maximum number of Cakes he can buy.

EXPLANATION:

This problem follows the Sliding Window pattern and is quite similar to Longest Substring with K Distinct Characters. In this problem, we need to find the length of the longest subarray with no more than two distinct integers (or cake types!). This transforms the current problem into Longest Substring with K Distinct Characters where K=2.

PROBLEM:

Meena and **Vaibhav** are suspects in a murder case and are bought in front of a judge.

The judge has his way of giving justice. Whenever the case involves two suspects, he announces his verdict based on a game. He gives a binary string S of length N to the suspects. A binary string consists only of "1"s and "0"s. Each suspect plays in alternate turns. In every turn the player must choose any index 1≤i≤N1≤i≤N such that Si≠Si+1Si≠Si+1 and delete SiorSi+1SiorSi+1 from the string. The player who is unable to make his move loses the game.

Determine the criminal if Meena plays first.

EXPLANATION:

First, count the number of 0s and 1s in the given binary string. If any of them is 0 then the answer is Meena. If the minimum of them is 1, then answer is Vaibhav. After covering these base cases if N is even then output is Meena else Vaibhav.

PREREQUISITES:

Strings, palindromes, ad hoc, dynamic programming

PROBLEM:

Given N, output a string consisting of lowercase letters such that there are exactly N palindromic substrings. Two substrings are considered different if they have different starting / ending positions.

QUICK EXPLANATION:

Print a string of length N of the following form: `abcabcabcabcabcabc...`. This string has exactly N palindromic substrings.

EXPLANATION:

This problem admits many kinds of solutions. We'll first describe the simplest one.

Simple

Notice that any string with length K has at least K palindromic substrings, because every single-character substring is palindromic! Thus, the answer, if it exists, must have a length of $\leq N$. Unfortunately, most strings of length K actually has *strictly more than K palindromic substrings*. For example, `abababab` contains 8 one-character palindromic substrings, but it also has the substrings `aba`, `bab`, `ababa`, etc., appearing multiple times. In fact, if there is a character appearing twice consecutively, then you automatically have more than K palindromic substrings. In the extreme case, in a string consisting of only one letter like `aaaaa...`, all substrings are palindromic.

But in fact there are ways to construct a string of length K having exactly K palindromic substrings. One of the simplest is a string of the form: `abcabcabcabcabcabc...`. It's easy to see why it doesn't have any palindromic substring of length > 1: any substring of length > 1 has at least one of the following substrings: `ab`, `bc`, and `ca`. But the reversals of these substrings don't appear anywhere in the string!

Thus, a very, very simple solution arises: Simply print a string of the form `abcabcabcabcabcabc...` of length N!

Short

The above solution is probably the simplest you can ever get. But what if we are conserving letters, i.e., we want the string to be as short as possible?

In that case, we want our string to actually have lots of palindromic substrings of length > 1. As mentioned above, an example of this is a string consisting of only one character like `aaaaa...`. If the length is K, then there are $1 + 2 + 3 + \ldots + K = K(K+1)/2$ palindromic substrings. Thus, to form a string with N palindromic substrings, we simply find a K such that $K(K+1)/2 = N$. Unfortunately, not all N s can be represented as $K(K+1)/2$. In fact, such numbers are rare! Below x, there are only approximately $\sqrt{2x}$ such values.

The idea in this solution is to append such single-character strings together to form one large string. For example, consider the strings `aaa...aaabbb...bbb`. Suppose there are A as and B bs. It's easy to see that no palindromic substrings containing an `a` and a `b` because if so, the substring `ab` would appear, but the reversal, `ba`, doesn't appear anywhere in the string! Thus, there are $A(A+1)/2 + B(B+1)/2$ palindromic substrings.
Similarly, `aaa...aaabbb...bbbccc...ccc` has $A(A+1)/2 + B(B+1)/2 + C(C+1)/2$ substrings, etc...

So our goal is to find a representation of N as a sum of *triangle numbers* 25 where the sum of the bases is as small as possible. We can achieve this using dynamic programming: Let $F(N)$ be the smallest sum of bases for any representation of N as a sum of triangle numbers. Then a simple recurrence can be derived by enumerating all possibities for the *last summand*. Specifically, if $k(k+1)/2$ is the last summand in the representation, then:

$$F(N) = \min_{\frac{k(k+1)}{2} \leq N} (k + F\,(N - \frac{k(k+1)}{2}))$$

Our base case here is simply $F(0)$. Along with $F(N)$, we also store the k that minimizes the above, so that we can reconstruct the representation!

The following pseudocode illustrates this:

```
BOUND = 10011
F[0..BOUND]
K[0..BOUND]

for N = 1..BOUND:
    F[N] = INFINITY
    for k=0 increasing where k*(k+1)/2 <= N:
        cost = k + F[N - k*(k+1)/2]
        if F[N] > cost:
            F[N] = cost
            K[N] = k // store the k that minimizes
```

Now, to find the representation of some N, we simply use the `K` array:

```
def find_representation(N):
    rep = []
    while N > 0:
        k = K[N]
        rep.append(k)
        N -= k*(k+1)/2
    return rep
```

Finally, we can construct the string itself using the representation with something like this:

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'
def construct(N):
    rep = find_representation(N)
    s = ''
    for i=0..rep.length-1:
        append alphabet[i] to s rep[i] times
    return s
```

With this answer, it turns out that you only need a string of at most 161 characters, and at most 5 distinct characters!

Investigating the shortest possible strings

The algorithm above produces some really short strings. Unfortunately, they aren't the shortest possible. The smallest counterexample is $N = 26$: The algorithm produces a string of length 10: `aaaaaabbcd`. But in fact a shorter string is possible: `aaaaaabaa`.

In fact, it doesn't seem easy to figure out the shortest possible string. So unfortunately I don't have the answer for that 🙁 If you wish to investigate it, here are the optimal answers for every $N \leq 55$, along with an optimal string. Perhaps you can find a pattern and then derive a formula for the optimal string.

```
 N opt string
 1   1 a
 2   2 ab
 3   2 aa
 4   3 aab
 5   4 aabc
 6   3 aaa
 7   4 aaab
 8   5 aaabc
 9   5 aaaba
10   4 aaaa
11   5 aaaab
12   6 aaaabc
13   6 aaaaba
14   7 aaaabac
15   5 aaaaa
16   6 aaaaab
17   7 aaaaabc
18   7 aaaaaba
19   8 aaaaabac
20   8 aaaaabab
21   6 aaaaaa
22   7 aaaaaab
23   8 aaaaaabc
24   8 aaaaaaba
25   9 aaaaaabac
26   9 aaaaaabab
27   9 aaaaaabaa
28   7 aaaaaaa
29   8 aaaaaaab
```

Time Complexity:

$O(N)$ 7

PROBLEM:

Given a string consisting of as and bs, what is the minimum number of letters to be changed so that all letters are the same?

EXPLANATION:

Count the as and bs, and print the smaller count. (There's really nothing more to it.) To make all letters the same, they will either be all as or all bs, so either all bs or all as need to be changed. Since we want the minimum, the answer is the smaller one.

There are only very few non-AC verdicts for this problem, and the only mistakes are (1) compile errors and (2) failing to initialize the array (to contain the string) properly.

Compile errors are some of the most unproductive verdicts in a contest, because they can easily be discovered and fixed (by simply compiling the code!). Though there are no penalty points for compile errors, remember that this is a short contest, so saving a few units of time and effort helps.

Failing to initialize the array properly can happen in one of the following ways:

- By failing to initialize the array at all.
- By failing to allocate an array large enough to contain the input. Remember to look at the constraints to know how much you need to allocate. Remember that in C++, you need an array of length $n+1$ to store a string of length n, because strings should be terminated by a null character ('`\0`').

This is mostly a C/C++ problem though. The simplest ways to take a string input in Java don't involve allocating an array yourself, so this problem doesn't happen.

This is the easiest problem in the contest, so you should try to spend little time to get this one AC'ed, to give you more time to solve the others.

Time Complexity:

$O(n)$

PROBLEM:

Given N dishes each represented by a string of lowercase characters, each character representing a different ingredient, find out the number of special ingredients. A special ingredient is an ingredient which is present in all dishes.

QUICK EXPLANATION

- Initially, all characters are considered special.
- For every string, we can mark the characters not present in the string as non-special. Hence, the characters marked special after considering all strings are the required set of characters.

EXPLANATION

The problem is really simple. We just need to check for each character from ‘a’ to ‘z’ whether this character is present in all strings or not.

This can be done in many ways, such as

- For each character, iterate over all strings and check if this character is present in all strings. If yes, increment the answer by one. The final value of the answer is the number of special ingredients.
- Maintain a special integer array of size A being the size of the alphabet, and for every unique occurrence of a character in the string, increase the character position in the array by one. The final answer is the number of characters, which have special array value equal to N.

The method which can be a learning experience is, by using bitmasks. Let us represent each character by a bit. Initially, all bits are set. Now, we represent each string as a bitmask, corresponding bit on for each character present in the string. Can you figure out the bitwise operation required here?

Click to view

We need to preserve only the bits which are set in both masks. This is what AND operation does.

Time Complexity

Time complexity is $O(|S|)$ per test case.

PROBLEM:

Given two integers N, K and a binary string S, for each character having **0** at index i, find the nearest index j having character **1** and find the sum $|i-j|+k$ for all the valid i.

EXPLANATION:

For each i, find the nearest **1** to its left or if not found then assign the value **INT_MAX**, let's denote it as $L(i)$ and find the nearest **1** to its right, let's denote it as $R(i)$ or if not found then assign the value **INT_MAX**. Let's denote the final answer as ans then find the minimum of $L(i)$ and $R(i)$ and add it to ans and add K for every i. And at last print the value of ans. Which can be done in $O(N)$ i.e. Linear pass.

PROBLEM:

Given 2 strings of length N represents two groups and determines which group moves to the next level.

QUICK EXPLANATION:

Traverse the string in any one direction. Determine whether the group follows both the given condition or not and output according to it.

EXPLANATION:

As per the given problem, the necessary condition for a group to move to the next level is,
(1) There must be an equal number of male (m) and female (f) players.
(2) Male and female players must have an alternate position.
To satisfy condition (1) N always have to be Even because that's one of the way that there are equal no of male and female players and to check
condition (2) we traverse through a string in any one direction and check $s[i]! = s[i+1]$ where s is a string and i is the index of characters.

Output "$first$" if group 1 wins, "$second$" if group 2 wins, "tie" if both groups satisfy the condition, and "-1" if both groups don't.

Time Complexity: O(n) per test case, where n is the length of a string.

PROBLEM:

Print the index of the leftmost largest contiguous binary substring consisting of only $1s$.

QUICK EXPLANATION:

Traverse the string from left to right and keep a variable for counting the number of contiguous $1s$ in the binary string. Reset the variable every time you encounter a 0. Keep a counter for assigning 1 based indexes to the contiguous $1s$. Print the minimum such index.

EXPLANATION:

As per the problem, Lelouch buys the largest chocolate bar. If there are multiple such chocolate bars then he buys the leftmost such chocolate bar. The chocolate bars in this problem statement correspond to the largest substrings consisting of only $1s$ in them.

Since we need to print the index of chocolate bars, we need to assign them indexes starting with 1 to the leftmost chocolate bar. In order to do that we must keep a $counter$ that can be incremented as soon as we reach the end of a chocolate bar. We also need to count the length contiguous $1s$, for which we will keep a variable named $curlen$, which has the length of current chocolate bar that you are traversing.

Each time we encounter a larger chocolate bar than the previous one, we need to update another variable named $totlen$ which stores the length of largest chocolate bar that has yet been encountered, and also we need to update ans which stores the index of largest-leftmost chocolate. We know that a chocolate bar ends when the next character in the string is a 0, the only exception is the last chocolate bar, for which we can append a 0 at the end of the string to make the implementation a bit simpler.

Finally just print the value of ans.

Time Complexity: O(n) per test case, where n is the length of string.

PROBLEM:

You are given two **nice strings** - G and M. Two strings are **nice**, if the letters are the same but the order is different.

We have to convert G to M. The only operation allowed is to pick any character from G and insert it at front.

Find the **minimum number of steps** required to convert string G to M.

NOTE : In one step, one letter of string G is added at the front of string G. In every subsequent step, the string formed in the previous step is used for operation.

EXPLANATION:

Let us suppose that length of G = length of M = N.

For transforming G to M, start matching from last characters of both strings. If last characters match, then our task reduces to $N-1$ characters. If last characters don't match, then find the position of $M's$ mis-matching character in G.

The **difference** between two positions indicates that these many characters of G must be moved before current character of G.

Below is complete algorithm –

1. Initialize $steps$ as 0
2. Start traversing from end of both strings
   - If current characters of G and M **match**, i.e., $G[i] = M[j]$, then do $i = i - 1$ and $j = j - 1$
   - If current characters **don't match**, then search $M[j]$ in remaining G.
     While searching, keep incrementing $steps$ as these characters must be moved ahead for G to M transformation.

SOLUTIONS:

Setter's Solution (CPP)
Tester's Solution (Python)

Feel free to share your approach. In case of any doubt or anything is unclear, please ask it in the comments section. Any suggestions are welcome

PROBLEM:

Given a string S find the frequency of each character in the string and check whether they can be rearranged into a sequence F where $F_i = F_{i-2} + F_{i-1}$ holds for all $i \geq 3$.

EXPLANATION:

Finding the frequency of each character can be done in linear time. One possible way is below

```
m = empty map
for each character c in S:
    if c in m:
        m[c] = m[c] + 1
    else:
        m[c] = 0
F = empty list
for each key, value in m:
    append value to F
```

Next we can say that because $F_i = F_{i-1} + F_{i-2}$ and F_{i-2} cannot be 0, $F_i > F_{i-1}$ for all $i \geq 3$. So it makes sense to sort the array F.

Then we can check if F satisfies the given condition for all $i \geq 3$. If it does, then the string is dynamic otherwise it is not, right? …But hold on, there is a catch. Indeed $F_i > F_{i-1}$ for all $i \geq 3$, but what about F_2? The relation between F_2 and F_1 is not specified. So it maybe that $F_4 \neq F_2 + F_3$ in the sorted order but $F_4 = F_1 + F_3$. In that case if we can simply swap F_1 and F_2 to get the required sequence and the string is dynamic.

For example: $F = (1, 2, 3, 4)$. Here $3 = 1 + 2$ but of course $4 \neq 2 + 3$. If we swap 1 and 2 we will get $(2, 1, 3, 4)$ where $3 = 2 + 1$ and $4 = 1 + 3$.

```
sort F
N = length of F
if N >= 4 and F[4] != F[2] + F[3]:
    swap(F[1], F[2])
ok = True
if N >= 3:
    for i in [3..N]:
        if F[i] != F[i - 1] + F[i - 2]:
            ok = False
if ok:
    S is dynamic
else:
    S is not dynamic
```

AUTHOR'S AND TESTER'S SOLUTION:

Author's solution can be found here 186

Tester's solution can be found here 125 .

PROBLEM:

In this problem, we need to find the number of ways to split a string S to substrings such that if there are k substrings ($p_1, p_2, p_3, \ldots, p_k$) in partition, then $p_i = p_{k-i+1}$ for all i ($1 \le i \le k$) and k is even.

EXPLANATION:

Let n be the length of the string s. Consider the string t = s[0]s[n - 1]s[1]s[n - 2]s[2]s[n - 3]…s[n / 2 - 1]s[n / 2]. The problem can be reduced to finding the number of ways to partition string t into palindromic substrings of even length.

Proof: Let k be the total number of partitions. Let $p_i = p_{k-i+1} = x_1x_2x_3\ldots x_m$ where m denotes length of p_i and x_j denotes jth character of p_i. The part of string t corresponding to these two partitions is $x_1x_mx_2x_{m-1}\ldots x_{m-1}x_2x_mx_1$ which is an even length palindrome. Similarly, the converse is also true.

Dynamic programming can be used to solve the problem. Let dp[i] be the number of ways to partition t[1…i] into even length palindromes. Then, where t[j + 1…i] is an even length palindrome. Of course for odd i, dp[i] = 0.

As discussed in this blog 26 , we can use an eertree to implement the solution. On the other hand, we can avoid the use of any suffix structure by following the algorithm described in this paper 12 .

Complexity: O(|s|log|s|)

Pre-requisites : Strings, KMP

Problem : Given two String A and B of the same length. You can do some (may be none) shift operations which results in the first character of B moving to the end. What is the minimum number of operations that are needed so that B has the longest common prefix with A.

Explanation

At first, let's consider some partial solutions.

How to get 30 points

Just try implement the shift operations in O(N) and try all possible numbers of Shift operations on B. You properly even do not need write so much code if your language have some built-in library for string. You also do not have to do anything to speed up the string comparison. The overall complexity is $O(N^2)$.

How to get 60 points

Let's say we append string B at the end of string B. We get: $B_1,B_2\ldots B_N B_1,B_2\ldots B_N$. The thing worth noting is that after K shift operations we get $B_{K+1},B_{K+2}\ldots B_N,B_1,B_2\ldots B_K$. This string(which we get after K shift operations) is now a substring of the new string that we got by appending B at the end of B.
So we'll now search all prefixes of A in B.B(means concatanation of B with B) which will help us in knowing which prefixes of A are present in B.B.
Let's say we have to search a string A of length M in a string B of length N. What we can do is generate hash of all substrings of B which are of length M and search for hash of string A in the generated hashes.
A hash function of string S could be say Summation[i=0 to length][26^i*(S[i]-'a')] modulo some prime P. If we have the hash of a substring of length N say S[i,i+N], we can easily generate hash of S[i+1,i+N+1] by using inverse modulo since we are using a prime for modulo operations.

How to get 100 points

You need to know string matching algorithm KMP to get full points. You can learn KMP here 130 , here 51 , or anywhere you like to :).
KMP helps us in finding all prefixes of a string X in string Y. So we'll search A in B.B(means concatanation of B with B) which will help us in knowing which prefixes of A are present in B.B. We'll choose an answer that gives us the largest prefix, also it'll give the index at which that prefix matches, from which we can find the number of shifts required.

Solutions : setter 193 tester 94

PREREQUISITES:

Tries, Hashing, KMP / Z algorithm

PROBLEM:

You're given a string **S** of length **N**. You create its suffix trie and then remove all edge labels. What you get is a directed acyclic graph. Let's call this as DAG of string **S**. Your task is to find out how many strings of length N are there whose DAG is isomorphic to DAG of **S**.

QUICK EXPLANATION:

Following invariants have to be preserved :

1. Number of leaves in trie
2. Length of longest common prefix (LCP) of any 2 suffixes in trie.

DETAILED EXPLANATION:

This was the hardest problem of the contest set by Gennady. It's an awefully difficult problem so don't be disappointed if you don't understand this editorial fully in first go.

We'd call any other string whose suffix tree is equivalent to **S**'s suffix tree as good string and through out deal in 1-based strings. Now let's start making some observations :

- Number of leaves in suffix trie depend upon length of the longest suffix of **S** which is also a prefix of some other suffix of **S**. Further more if length of this suffix (denoted as **X** from now on) is **L**, number of leaves in suffix trie is **N - L.** :

Let's start adding suffixes to our trie one by one starting from the longest one. Uptil some point, for each new suffix we get a new leaf in our trie. Now consider that first suffix which when added to trie creates no new leaf. This suffix couldnt have created any node either as any suffix which creates new node creates a leaf. So it means this whole suffix was present as path from root already. Hence it was prefix of some suffix which was already added. Any suffix that'd be added after this won't create any new leaves either.
Further more, if length of the first suffix that dint create a leaf was **L**, we had already added **N-L** suffixes creating **N-L** leaves. So if **L** is the length of the longest suffix which is also the prefix of some other suffix, there are exactly **N-L** leaves in the tree.

- Number of distinct letters in any good string is constant.

Degree of root of the tree is equal to the number of distinct letters in the string. As for isomorphic trees, degrees are also preserved, degree of root is also preserved and hence all good strings have same number of different letters.

- All distinct letters in any good string would've appeared atleast once in its first **N-L** letters.

If possible assume this is not the case : there is a good string for which there is some letter that occurs for the first time after **N-L** letters.

S----- (N - L) -----------> X---- L —>
P---- L —>

S----- (N - L) -----------> X---- L —>
P---- L —>

These are the two cases. In first case clearly all letters of **X** have appeared in first **N-L** characters. In second case, take the first occurence of character that appears first time in **X**. This same character also must appear in portion of **P** which contains **X** and hence appears to the left as well but this is contradiction as we chose the first occurrence of that letter.

- For all **1 <= i, j <= N-L**, if **i^{th}** and **j^{th}** letters were same in **S**, they would be same in all good strings and if they were different in **S**, they'd be different in all good strings.

[a]Assume they were same in **S**. Then look at paths of length **N-i+1** and **N-j+1** from leaf to root in tree. Such paths exist and are unique - they belong to **i^{th}** suffix and **j^{th}** suffix respectively. If **i^{th}** and **j^{th}** letter were same in **S**, these two paths share atleast 1 edge. As all trees are isomorphic, they all have
this property. Also it is easy to see that if a tree has this property, then **i^{th}** and **j^{th}** letters have to be same. Hence the claim.

- First **N-L** letters of any good string are uniquely determined modulo some permutation.

From previous three claims, all good strings contain exactly **D** distinct letters which have appeared in first **N-L** characters. Also we know that which characters are equal amongst first **N-L** characters and which are not. This way we can partition first **N-L** character into **D** groups - where each group has to get a distinct letter. We can assign letters to these groups in **26 * 25 * … (26-D+1)** ways. Modulo this permutation, all letters have been fixed.

From now on, we will assume that these **N-L** letters are same as first **N-L** letters of **S** and would later multiply our answer by **26 * 25 … (26-D+1).**

- There are at most **N-L** ways of filling remaining **L** characters.

We know that last **L** character (also called as suffix **X** in discussion) have to be prefix of some other longer suffix. There are **N-L** longer suffixes. As soon as we fix a longer suffix (denoted by **P** in subsequent discussion), all the remaining **L** characters are uniquely determined (letter by letter, starting from the first one). Not all of these **N-L** different candidates of **P** however
give us a solution so there are at most **N-L** different trees.

- [b] For all **i <= N-L**, longest common prefix (denoted as LCP from now on) of **i^{th}** suffix and **N-L^{th}** suffix (denoted as **LCP(i, N-L)**) is an invariant.

Actually from explanation [a] of a previous claim, it is easy to see that **LCP(i, j)** is preserved for all **1 <= i, j <= N-L**. Now as we've already arranged first **N-L** characters so that pairwise equality of letters is maintained - we can move both the indices forward until one of them becomes **N-L**. More formally, if **LCP(i, N-L)** is preserved for all **i <= N-L**, we can be assured that **LCP(i, j)** is preserved for all **i,j <= N-L.**

Approach 1:

So now we have an **O(N^2)** solution based on observations made already. Fix some **i <= N-L** such that ith suffix is our **P**. There are **O(N)** such different **P**. For a given **P**, find out remaining characters in **O(N)** time using simulation. After that run Z algorithm to find out **LCP(i, N-L)** for all **i** in our generated string in **O(N)** time. If all of these are same as corresponding values of **S**, we've found a new way. A subtle point here is to note that different candidates of **P** might give us

same **X**. To avoid counting them more than once, we put all different **X** in a set and finally take its size. One could also put hashes of these **X** and put them in set instead of putting whole **X** itself.

Finally ans is = **|# of ways of picking P that give distinct X| * 26 * 25 … * (26-D + 1).**
But as this takes **O(N^2)** time, it is not sufficient to get Ac. We need to do make more observations now.

- Denote **Q** = max over **i <= N-L** { **LCP(i, N-L)** } and let **K**th suffix be one such suffix which achieves this values of **Q**. First **Q-1** letters of X are uniquely determined.

As from a previous claim[b], **LCP(K, N-L)** is to be preserved across all good strings, and it is **Q** in **S** - it has to be **Q** in all good strings. That means we can fill up **Q-1** characters after **N-L** letters based on next **Q-1** characters of **K+1**th suffix.

So now we can assume that we know exactly **N-L+ Q-1** characters of any good string uniquely. We've to try to fill in remaining characters.

- No matter how we choose remaining characters, for all **i <= N-L**, **LCP(i, N-L)** in our string >= **LCP (i, N-L)** in **S**

Choose any **i <= N-L**. There are two cases:
Case 1 : **LCP(i, N-L) < Q** in **S**. In this case it is easy to see that **LCP(i, N-L)** in our string == **LCP(i, N-L)** in **S**.

Case 2 : **LCP(i, N-L) == Q** in **S**. In this case we've already ensured that **LCP(i,N-L)** in our string >= **Q**.

From this it is also clear that we have still to ensure that for all **i <= N-L, LCP(i, N-L)** in our string == **LCP(i, N-L)** in **S**. For this we must remember following rule: if for some **i <= N-L, LCP(i, N-L) = Q**, then **Q**th character of suffix **X** should be different from **Q+1**th character of suffix **i**. So now we've set of forbidden letters which can't come in **Q**th position of **X**.

- We can examine all candidates of **P** in **O(1)** time each.

For any candidate of **P**, we've to check that its first **Q-1** letters are same as those of first **Q-1** letters of **X** that we've already fixed. This we can do either using hashing or using string algorithms like Z or KMP. Then we must check that **Q**th letter of **P** is not a forbidden letter. This is again **O(1)** check.

However one problem with this approach is that multiple candidates of **P** might give same suffix **X**. All such candidates of **P** should be counted only once.

- We can use hashing to find out number of distinct **X** we can get in total in time **O(N log N)**

Our plan is to find out hash of all distinct **X** that we can obtain and put them all in a set so as we can find out only unique strings. So now the only problem is to find out hashes quickly.

For all **i <= N- 2L**, we can find out hash of next **L** characters in constant time after precomputing sequence hash on **S**. For all **i > N-2L**, our **X** would contain some full periods of **S[i…N-L]** and one partial (probably empty) period. Length of this period denoted by **len** is equal to **N-L-i+1**. Number of full periods is **L / len** and partial length is **L % len**.

If we deal with only polynomial hashes, we can find out hash of one period in constant time (it is nothing but hash of **S[i…N-L]**). If hash of string **A** is **H**, then hash of **AA** (two copies of **A** is) : **H * PRIME**$^{len(A)}$ **+ H**.

Continuing this way we can find out hash of **X** in time **O(L/len)**. Total time taken is : summation over **len { L / len }** which is **O(L logL)**. As **L <= N**, Total time is **O(N log N)**.

I'm repeating with emphasis, this is a fairly invovled problem and you should read editorial again and again if you don't get it and try out lot of examples by hand. You can also look at Tester/Setter solution for reference. You could also take a look at setter's **O(N^3)** solution 1 for a clearer understanding.

PREREQUISITES:

Strings, Greedy algorithms

PROBLEM:

A string is K-good if for every two letters in the string, if the first appears x times and the second y times, then $|x - y| \leq K$. Given K and a string w, what is the minimum number of letters to remove to make w K-good?

QUICK EXPLANATION:

For every m, define $F[m]$ as follows:

- Remove every letter that appears $< m$ times.
- For every letter that appears $> m + K$ times, remove just enough letters so that it appears exactly $m + K$ times.
- Let the number of removed letters be $F[m]$.

The answer is then the smallest among $\{F[1], F[2], \ldots, F[N]\}$. To compute this quickly, simply precompute the letter frequencies in w beforehand, and operate on that array when computing $F[m]$.

EXPLANATION:

Let's use the notation $w_{(c)}$ to denote the number of appearances of the letter c in the string w.

K-goodness

By definition, a string is K-good if for every two characters c and d, $\left|w_{(c)} - w_{(d)}\right| \leq K$. But for a given string w, the value of $\left|w_{(c)} - w_{(d)}\right|$ is maximized if c and d are the most and least occurring characters in w (or vice versa). Thus, we have a potentially simpler criterion for telling whether a string is K-good:

If c and d are the most- and least- occurring characters in w, then w is K-good if and only if $w_{(c)} - w_{(d)} \leq K$.

Notice that we have removed the absolute value because $w_{(c)} \geq w_{(d)}$.

Removing letters

This immediately gives us a **greedy** algorithm for turning w into a K-good string in a few letter removals:

- Compute the frequency counts for each of the 26 letters.
- Find the least-occurring character d. (Note that we only consider characters that actually appear in w, that is, whose frequency counts are positive.)
- For every character c appearing more than $w_{(d)} + K$ times, remove $w_{(c)} - (w_{(d)} + K)$ instances of c.

Clearly this algorithm yields a K-good string if you use our criterion above. Unfortunately though, *it doesn't yield the minimum number* of removals! For example, consider the input `boooooooooo 0`. The algorithm above will have us remove 9 instances of the letter 'o', yielding the string "bo" which is 0-good. But it's possible to do it with just 1 removal: just remove the 'b'!

So how do we fix this? Clearly, we can't assume that the least-occurring character in w will appear in the final string. In fact, it isn't obvious what the minimum frequency count must be in the final string. So what do we do? We just *try*

them all!

Our adjusted algorithm would be:

- Decide on the minimum frequency count on the final string, and call it m.
- Remove every letter that appears less than m times.
- For every letter c that appears more than $m + K$ times, remove $w_{(c)} - (m + K)$ instances of c.

The answer is then the minimum number of removals for all such values m in the range $m \in [1, N]$. Now, clearly this is correct, but is it fast enough? Well, it depends on your implementation. Obviously, if you process the string for every m, you'll get an $O(N^2)$ running time which is pretty bad. But thankfully, we can reduce it by noticing that *every iteration of the above procedure only requires the frequency counts of all the letters!* By computing the frequency counts just once and then using that array instead, we get a running time of $O(N)$!

Here's a pseudocode:

```
// compute counts
counts['a'..'z'] // initialized to zero
for i=0..|w|-1:
    counts[w[i]]++

// compute answer
answer = N
for m=1..N:
    current = 0
    for c='a'..'z':
        if counts[c] < m:
            current += counts[c]
        else if counts[c] > m + K:
            current += counts[c] - (m + K)

    if answer > current:
        answer = current

print answer
```

Optimization

We can actually still optimize the algorithm above, by noticing that we don't have to try all values m in the range $[1, N]$. Instead, we only try the m values that are equal to some $w_{(c)}$!

Why is that? Suppose we chose a value m and it's not equal to any $w_{(c)}$. Then if we instead choose $m + 1$, then there aren't any additional characters whose frequency counts are $< m + 1$, so we don't have to remove any additional characters. However, we potentially save a few removals, particularly those whose frequency counts are $> m + K$! Thus, the result of the procedure above for m cannot be strictly less than its result for $m + 1$, so it follows we don't have to check m if it's not equal to any $w_{(c)}$!

With this optimization, the number of times we perform the procedure reduces from N to (at most) 26! To formalize the optimization, notice that our original $O(N)$ algorithm actually runs in $O(N\alpha)$ where α is the size of the alphabet. This optimization reduces that to $O(N + \alpha^2)$, which is usually better because $N \gg \alpha$ usually!

Here's a pseudocode:

```
// compute counts
counts['a'..'z'] // initialized to zero
for i=0..|w|-1:
    counts[w[i]]++

// compute answer
answer = N
for d='a'..'z':
    m = counts[d] // consider only the relevant m's
    current = 0
    for c='a'..'z':
        if counts[c] < m:
            current += counts[c]
        else if counts[c] > m + K:
            current += counts[c] - (m + K)

    if answer > current:
        answer = current

print answer
```

Time Complexity:

$O(N)$

PROBLEM:

You are given a string S (consisting of lowercase Latin letters only) and you can rearrange the letters of this string in any way. You have to find out if it is possible to rearrange the letters of string S to obtain a Palpal string.

A string S (consisting of lowercase Latin letters only) is called a Palpal string if it can be divided into contiguous substrings such that:

- Every Character of the string is present in one and only one substring.
- Every Substring is a palindrome of $length \geq 2$

EXPLANATION:

Our task is to divide the string into contiguous substring of length greater than or equal to two, such that every substring is a palindrome.

If there is a palindromic substring and we delete the leftmost and the rightmost character from this substring, the resulting substring will remain palindromic. It means to create a palindromic substring of length x, we need to create a palindromic substring of a smaller length too. Hence the most optimal way to divide the string is in the smallest length possible which is 2.

Dividing the string S into substrings of length 2 will also increase palindromic substrings which will be useful when we need to insert the odd character in these substrings as explained later.

Now we will try to form palindromic substrings of length 2. We can see that, if any letter is present an odd number of times in this string then a single character of this letter won't be able to form a palindromic substring of length 2. But all such characters can be inserted in between substring of length 2 since the addition of this character won't make the substring non-palindromic. It means that there should be enough substrings such that all such characters could be inserted.

Let us suppose the count of such characters is y. Now if the number of substrings of length 2 is more than or equal to y, then it is always possible to insert such characters in between the substrings making the substring still palindromic. If such conditions satisfy then it is possible to rearrange the letters of string S to obtain a Palpal string.

TIME COMPLEXITY:

$O(|S|)$ per test case.

PREREQUISITES:

Strings, observation, basic math

PROBLEM:

Beeds Arranging game is described by a string of length nn consisting of characters "." (empty space) and "*" (beed). In one move you can shift beed to one unit left or one unit right, only if the corresponding place *exists and is empty*. The game ends up as soon as the beeds are lined up, that is, there should not exist any empty cells between any beeds.
FInd minimum number of moves.

EXPLANATION:

Let's denote k the number of beads in the string, and $x_q, x_2, ..., x_k$ $(1 \leq x_1 < x_2 < ... < x_k \leq n)$ their positions in the string.
Note that in the optimal solution the beed with number $m = ceil(n/2)$ will not make moves. This can be proved by considering the optimal solution in which the beed with the number m makes at least one move and come to the conclusion that this solution is not optimal. Consider beed with numbers from $i = 1$ to n. Then the final position of the $i^{th} beed will be $x_m - m + i$, and the answer will be $\sum |xi - (x_m - m + i)|$ where i runs from 1 to k.

COMPLEXITIES:

Time Complexity: $O(N)$ for each test case

PREREQUISITES:

Comparator based Sorting

PROBLEM:

Arvind loves playing with Numbers. He has a sequence $[A_1, A_2, \ldots, A_N]$ of N integers. He wants to reduce the length of the sequence to 1. To do that, he performs exactly $N-1$ moves, where a move is defined as follows.

- Pick two Integers A_i and A_j, $i \neq j$, remove them from the sequence and append the result obtained by concatenating the Integers to the sequence.

He wonders what could be the Largest Integer he can obtain at the end. Can you help him in finding it?

QUICK EXPLANATION:

The problem can be easily solved by sorting the elements of the array based on a Comparator. Refer to the explanation for Comparator based Sorting.

Note: Since the numbers are very large (10^{10^3}), we read them into strings.

EXPLANATION:

Given an array A containing N integers, we need to output the largest number that can be obtained by concatenating these N integers.

This problem can be solved using Greedy Approach. Consider the array of Integers $[a_1, a_2, a_3, \ldots, a_n]$. We sort the sequence by comparing two elements at a time.
Say $(S_1,\ S_2)$ belong to the sequence. The two possible ways of ordering them are:

1. $(S_1,\ S_2)$ => (S_1 comes before S_2) after rearranging.
2. $(S_2,\ S_1)$ => (S_2 comes before S_1) after rearranging.

Which ordering is optimal?

We can decide this by comparing them the following way:
If $(S_1 + S_2) > (S_2 + S_1)$, then the ordering shall be $(S_1,\ S_2)$. Otherwise, the ordering shall be $(S_2,\ S_1)$.

We can prove that the ordering of elements in this way will always produce the largest Number.

PREREQUISITES:

Strings, Hashing

PROBLEM:

Mr. Seth loves to play with strings, so he provides you a string S of lowercase alphabets and a positive integer Q.

You have to process Q queries of following two types:

- $1\ L\ X$ - change the character at index L to X. (1-based indexing)
- $2\ K$ - print the Kth largest frequency character in string S, if it exists otherwise print -1.

Note: If there exists multiple K^{th} largest frequency characters then print the **lexicographically smallest** one.

EXPLANATION:

- Brute Force approach would be, iterate over every character in the string and store it's frequency and then sort the frequency array to find the Kth largest if exists. To update the character at any index also you will need to iterate over string.

  This approach would take overall Time Complexity of $O(q * n^3)$ and works for subtask 1.

- Optimised Approach would be to create a hash of characters to store frequency (only takes $O(26)$ extra space) and then find the distinct frequencies for K^{th} largest frequency if it exists. In this approach updation will take constant time.

  This approach would take overall Time Complexity of $O(n + q)$ and works for all subtasks.

Note - Sorting and extra operations would take $O(26 * log26)$ time. So we considered it as constant.

Prerequisites

Divide and Conquer, Strings, Sortings

Problem Statement

In a retro game, the winner wins if the two strings a and b of equal length are called equivalent. They should satisfy either of the two conditions:
1] They are the same.
2] If we split both the strings into two equals halves, then one of the following is correct:
a) a1 is equivalent to b1, and a2 is equivalent to b2
b) a1 is equivalent to b2, and a2 is equivalent to b1

You need to determine if they are equivalent.

Approach

Let us note that "equivalant" described in the statements is actually equivalence relation, it is reflexively, symmetrical and transitive. Let's find lexicographic minimal strings what is equivalent to first and to second given string. And then check if its are equals.
It is remain find the lexicographic minimal strings what is equivalent to given. For instance we can do it such a way:

```
String smallest(String s) {
    if (s.length() % 2 == 1) return s;
    String s1 = smallest(s.substring(0, s.length()/2));
    String s2 = smallest(s.substring(s.length()/2), s.length());
    if (s1 < s2) return s1 + s2;
    else return s2 + s1;
}
```

Every recursive call time works is O(n) and string split by two twice smaller strings. Therefore time of work this function is O(nlog(n)), where *n* is length of strings.

Solution

```
#include <bits/stdc++.h>
using namespace std;
string dac(string s){
   if(s.size()%2)return s;
   string a = dac(s.substr(0,s.size()/2));
   string b = dac(s.substr(s.size()/2));
   if(a < b) return a+b;
   return b + a;
}
int main()
{
   string s, t;
   cin>>s>>t;
   cout << ((dac(s) == dac(t)) ? "YES" : "NO");
   return 0;
}
```

PREREQUISITES:

String processing

PROBLEM:

Given two strings A and B, is it possible to choose some non empty strings s_1 and s_2 such that s_1 is a substring of A, s_2 is a substring of B, and $s_1 + s_2$ is a palindromic string?

QUICK EXPLANATION:

If A and B share a common letter, output `Yes`. Otherwise, output `No`.

EXPLANATION:

This problem can be solved with a few simple observations.

First, what is the simplest palindrome you can think of? Surely, a single letter word is a palindrome. However, such a palindrome cannot be obtained as $s_1 + s_2$ because s_1 and s_2 must be nonempty (which means the length of $s_1 + s_2$ is at least two). So the next best thing is a two-letter palindrome, which is just a letter repeated two times. The only way we can form such a palindrome as $s_1 + s_2$ is when s_1 and s_2 are the same single-letter word. Thus, if A and B share a common letter, then the answer must be `Yes`.

But what if A and B don't have a common letter? Well, it turns out that the answer is `No`. This is because in any palindrome, the first and last letter must be the same. So suppose we can form the palindrome $s_1 + s_2$. Since the first letter belongs to s_1 (and thus to A) and the last letter belongs to s_2 (and thus to B), it means that A and B must have a letter in common!

(The arguments above show that A and B sharing a common letter is both **necessary** and **sufficient** for the palindrome $s_1 + s_2$ to exist.)

Thus, our solution is really simple: The answer is `Yes` if A and B share a common letter, otherwise, the answer is `No`.

In the following, we will describe a few methods on how to compute the answer.

Finding a common letter: Method 1

The easiest way is to simply find two indices i and j such that $A[i] = B[j]$. The following C code demonstrates it:

```
#include <stdio.h>

char A[1111];
char B[1111];
int main() {
    int cases, cas, i, j, good;
    scanf("%d", &cases);
    for (cas = 1; cas <= cases; cas++) {
        scanf("%s%s", A, B);
        int good = 0;
        for (i = 0; A[i]; i++) {
            for (j = 0; B[j]; j++) {
                if (A[i] == B[j]) {
                    good = 1;
```

```
                    break;
                }
            }
            if (good) {
                break;
            }
        }
        puts(good ? "Yes" : "No");
    }
}
```

Each `for` loop iterates through every character of every string. If a common letter is found (`A[i] == B[j]`), then we mark good as `1` and break out of both loops (using two `break` statements).

C/C++ note: Even though the problem says A and B are only up to 1000 in length, you may need to make the char arrays `A` and `B` at least 1001 in length, because you need to allocate space for the null terminator [2] .

Finding a common letter: Method 2

The code above isn't actually the fastest way to compute the answer, because it checks all pairs of indices, and there can be up to 1000000 such pairs. To compute the answer faster, we can use the fact that all letters are lowercase letters.

One way would be to check which of the 26 lowercase letters appear in A, and which ones appear in B, and find a letter that appears in both. The following Java code shows one way to do it:

```
import java.util.Scanner;
public class Main {
    public static void main (String args[]) {
        Scanner sc = new Scanner(System.in);
        int cases = sc.nextInt();
        boolean aHas[] = new boolean[256];
        boolean bHas[] = new boolean[256];
        for (int cas = 1; cas <= cases; cas++) {
            String a = sc.next();
            String b = sc.next();
            boolean good = false;
            for (char c = 'a'; c <= 'z'; c++) {
                aHas[c] = bHas[c] = false;
            }
            for (int i = 0; i < a.length(); i++) aHas[a.charAt(i)] = true;
            for (int i = 0; i < b.length(); i++) bHas[b.charAt(i)] = true;
            for (char c = 'a'; c <= 'z'; c++) {
                if (aHas[c] && bHas[c]) {
                    good = true;
                    break;
                }
            }
            System.out.println(good ? "Yes" : "No");
        }
    }
}
```

It uses the `aHas` and `bHas` boolean arrays to store which letters appear in which string. The final loop finds a letter "`c`" such that `aHas[c]` and `bHas[c]` are simultaneously `true`.

A fancier way of doing the above would be to use a bitmask [10], or using an integer's bits as a boolean array. The i th bit will be *on* if and only if the i th lowercase letter is present in the string. Since there are only 26 characters, only 26 bits are needed, so our mask would be a value between 0 and $2^{26} - 1$. The following C++ code shows how to do it:

```
#include <iostream>
using namespace std;

int main() {
    int cases;
    cin >> cases;
    while (cases--) {
        string a, b;
        cin >> a >> b;
        int a_mask = 0, b_mask = 0;
        for (int i = 0; i < a.size(); i++) a_mask |= 1 << a[i] - 'a';
        for (int i = 0; i < b.size(); i++) b_mask |= 1 << b[i] - 'a';
        cout << (a_mask & b_mask ? "Yes" : "No") << endl;
    }
}
```

It uses the *bitwise AND* [4] operator to check if the two masks have a common *on* bit, which only happens if A and B share a common letter.

Finally, we can use sets to represent the set of letters of A and B. This way, A and B share a common letter if and only if the *intersection* of their letter sets is nonempty. The following Python code shows one way to do it:

```
cases = int(raw_input())
for cas in xrange(cases):
    a_set = set(raw_input().strip())
    b_set = set(raw_input().strip())
    print "Yes" if a_set & b_set else "No"
```

It uses the fact that Python sets use the operator & for set intersection. Another bonus with this solution is that it also works even if letters are not restricted to lowercase letters! Also, such solutions can be reduced to just one line (in Python at least):

```
for cas in xrange(input()): print "Yes" if set(raw_input()) & set(raw_input()) else "No"
```

Time Complexity:

$O(|A||B|)$ or $O(|A| + |B|)$

PREREQUISITES :

Strings, frequency array

PROBLEM :

Given a string s consisting of only lowercase characters. You can perform following operation exactly once:
→ Choose any character from the English alphabets and remove some (or all) occurrences of this alphabet from the given string.
What is the smallest length of the string that you can obtain ?

QUICK EXPLANATION :

To get the smallest final string just remove all the occurrences of a character that appears maximum times in the string.

EXPLANATION :

→ You are asked to choose a character and remove some or all occurrences of it from the string. As you need to find the smallest string after above operation, it is very clear that we always have to remove all the occurrences of that chosen character.

→ Now, which character to choose ? Let x be our chosen character. Let n be the length of original length of the string. So after removing all occurrences of x, the length of final string will be $(n - count(x))$, where $count(x)$ is the number of times x occurs in the string. We can clearly see for minimum final length, $count(x)$ should me maximum. So, choose a character which occurs maximum time in the given string.

→ How to choose x ? Main part of the question is to choose optimum character. For that let's make a frequency array $freq$[ ] of size of lowercase English letters. The size of the array will be 26. Now, iterate over the characters of given string and increase the count of that character by 1 in the frequency array by $freq$[s[i] - 'a']++. Read more about frequency array here .

Now, just iterate over the $freq$[ ] array and choose the maximum value.
Final answer will be (length of original string - max).

TIME COMPLEXITY :

Time complexity is O(|S| +A) per test case, where A=26, the size of $freq$[ ].

PREREQUISITES:

Manacher's Algorithm O(n)

PROBLEM:

Given a string in "compact notation"(see the problem), find the number of substrings that are palindrome.

EXPLANATION:

The problem was a straightforward application of Manacher's Algorithm. This Algo was originally proposed to find the longest palindrome in a string. However, for each "center" c, it finds the length of the longest palindrome "centered" at c. Therefore, we can use it to find the number of palindromes "centered" at c as well. Handling the fact that strings are given in compact notation is relatively straightforward.

Many elegant descriptions of Manacher's Algorithm can be found on internet 4

In very short, Manacher's Algorithm can be written as follows.

```
1 | int p[N+1], mx = 0, id = 0;
2 | // length of longest palindrome centred at i is 2 * p[i]-1.
3 | for (i = 1; i <= N; i++) {
4 |     p[i] = mx > i ? min(p[2 * id-i], mx-i) : 1;
5 |     while (s[i + p[i]] == s[i - p[i]]) p[i]++;
6 |     if (i + p[i] > mx) {
7 |         mx = i + p[i];
8 |         id = i;
9 |     }
10| }
```

```
Let the compacted string be (c1, k1), (c2, k2), … (cN, kN).
```

- Palindromes that contain only a single character repeated several times can be counted as:

Assuming reader understands Manacher's Algorithm, here is how to modify it for this problem:
Σ ki * (ki +1) / 2

- Palindromes that span over more than 1 contiguous segment of compressed string can be computed by also maintaining an array q, which stores the length of the longest "decompressed" palindrome centered at the ith segment.:

q[i] = $k_{i-p[i]+1} + k_{i-p[i]+2} \ldots + k_i + \ldots k_{i+p[i]-2} + k_{i+p[i]-1}$
There will also be minor changes like:

1. We need not put interleaving '#'es because center of every palindromic substring is the center of some "compressed" segment(unless it fully lies inside a segment).
2. In line no 5, compare the characters as well as the lengths of the segments.
3. 1. If the segments **i-p[i]** and **i+p[i] do not have same lengths, but have the same character**, then **q[i]** would need to adjusted by adding **2*min($k_{i-p[i]}$, $k_{i+p[i]}$)**.

The final Answer is
Σ ki * (ki +1) / 2 + ⌈q[i]/2⌉ - ⌈ki/2⌉
if you are having difficulty in understanding it please go through this_link 2 although the problem is not the same, the algorithm explained there.

PREREQUISITES:

Strings, STL, Implementation.

PROBLEM:

Given a list of n words, you have to calculate its magic factor with the last word you encountered of the same length. Value of a word is the sum of values of each letter of the word. (For eg: **a**=1, **b**=2, etc) and Magic factor of 2 words having value $v1$ & $v2$ is = $abs(v1 - v2)$.

QUICK EXPLANATION:

Maintain a map and calculate the magic factor for every word encountered if a word of the same length is present in map.

EXPLANATION:

We can maintain a hash map for string lengths.

Every time we encounter a string whose size is n_1, we will check if a string of size n_1 is already available or not.

If there is an available string, we will calculate the diff factor of both strings and clear the map key of that particular length.

If there is no available string, we will store the value of that length in the map and move on.

Finally, we will print the sum of all magic factors calculated.

Time Complexity : $O(nlog(n) + |s|)$ for each test case

The extra $log(n)$ factor can be reduced if we use vector or unordered_map instead of map.

PREREQUISITES:

Palindrome,
String Processing,
String Manipulation

PROBLEM:

A typical problem of string manipulation and use of palindromes.

QUICK EXPLANATION:

A predefined notation is used to map the strings and then,using the notation format the number of palindrome sub-strings in the string is found out .

EXPLANATION:

The problem demands an analytical approach and good understanding of palindromes. There are various ways to solve this problem, one of which is explained here -
The solution modules are aimed at solving different parts of the problem -
In one module(method), the string notation is first expanded to generate the actual string, so that the analysis of the situation becomes easier.This module generates the string for the solve() module to find out the sub-strings. This is done by using simple mathematics to analyse the length of the string expected, and then producing the output to the solve method.

```
void expand(int len)
{
        - - - -

    int expand;
    for(i=2;i(<)n;i++)
    {
        mirror=2*center-i;

    - - - -

        if(diff>0)
         //do-something
        if(expand==1)
            //do-something
    }

    - - - -
}
```

The second module then focuses on solving the palindrome mystery by comparing each possible sub-string to its reverse string.This module takes the output of the expand function , to evaluate the palindromes in the given string and at last return the number count of the palindromes discovered.

```
void solve()
{
        int len=strlen(str),i;
        for(i=0;i(<)len;i++)

    - - - -
```

```
        expand(len);

    - - - -

        sum[0]=arr[0];

    - - - -

        for(int i=0;i(<)len;i++)
        {
                //do-something
        }
        printf("%lld\n", ans);
}
```

The output from both the modules is combined in the main function to display the required output.

PREREQUISITES

Suffix Tree
Dynamic Programming
String Theory

PROBLEM

In the Seventh episode of the Third Season of Rick and Morty as Rick and Morty were preparing to go to Atlantis, they were interrupted by Rick and Morty's pair from the Citadel of Ricks looking for donations to the Citadel Redevelopment fund. Our Rick and Morty refute the possibility of donation with hostility and the pair left (pointing out that our Rick was the one to say no).

Imagine a scenario where our Rick chose to give them one more opportunity to Rick from Citadel after all Rick from Citadel looked precisely like him. So our Rick asked them a problem based on pairs, which Rick and Morty from Citadel couldn't tackle since they were busy collecting donations, you liked the idea of donations for a social cause and chose to help them to crack the problem.

They were given a string str, determine the total n number of ways to cut string str to substrings such that if there are n (n is always even) substrings S_i (1≤i≤n) such that $str = S_1S_2S_3S_4 \ldots\ldots..S_(n-1)S_n$, then $S_i = S_n - i + 1$ $S_i = S_n - i + 1$ for all $i(1 \leq i \leq n)$. Since the answer can be huge, calculate its modulo $10^9 + 7$.

EXPLANATION

Let n be the length of the string s. Consider the string $nstr = str[0]str[n-1]str[1]str[n-2]str[2]str[n-3]...str[n/2-1]str[n/2]$. The problem can be reduced to finding the number of ways to partition string $nstr$ into palindromic substrings of even length.

Proof: Let *k* be the total number of partitions. Let $S_i = S_{k-i+1}$ for $S_1S_2S_3...S_n$ where *m* denotes length of p_i and x_j denotes j_{th} character of p_i. The part of string str corresponding to these two partitions is $p_1p_mp_2p_{m-1}...p_{m-1}p_2p_mp_1$ and this an even length palindrome. Similarly, the converse is also true.

Dynamic programming can be used to solve the problem. Let *dp[i]* be the number of ways to partition *t[1…i]* into even length palindromes. Then, *dp[i]=sigma(dp[i-1])* where $nstr[j+1...i]$ is an even length palindrome. Of course for odd *i*, *dp[i]* = 0.

we can use an EER tree to implement the solution. Further, we can avoid the use of any suffix structure by following the algorithm described here https://arxiv.org/pdf/1403.2431.pdf .

To know more about suffix tree: Suffix Trees Tutorials & Notes | Data Structures | HackerEarth

PREREQUISITES:

Hashing, strings

PROBLEM STATEMENT:

Given a string s, convert it into a string with not more than k distinct character, where the cost of changing a character x to character y is abs(ASCII(x) -ASCII(y)). The string is composed of only the first 16 alphabets.

EXPLANATION:

The problem makes use of the fact that there are only 16 valid alphabets.

->So first we preprocess a bitmask of 16 bits.
We store these bitmasks in a vector according to the number of setbits in each number.
For example, numbers like 1,2,4 etc with only 1 set bit in their binary representation are stored in a single vector.
Whereas numbers like 3, which have 2 set bits are stored in a separate vector.

→ Next we create a frequency table of all the letters appearing in the string s.

→ Finally, given k, we try each mask which has number of set bits equal to k.
Here, ith bit of the mask being 1 corresponds to the ith alphabet (with a=0,b=1,c=2,…,p=16),wherever it appears in s, is also present in the final string p.
Similarly, the ith bit of the mask being 0 indicates that the ith alphabet, wherever it appears in s, is changed to some other alphabet in final string p.

The alphabets which correspond to 0 bits in the mask, are changed to the nearest alphabet which has value 1 in the mask.
The cost is calculated for each mask in this way, and then the minimum cost is taken as the final answer.

PROBLEM:

You are given a matrix of characters. The matrix has *N* rows and *M* columns. Given a string *s*, you have to tell if it is possible to generate that string from given matrix.
Rules for generating string from matrix are:

1. You have to pick first character of string from row *1*, second character from row *2* and so on. The (N+1)th character of string is to be picked from row *1*, that is, you can traverse the rows in a cyclic manner (row *1* comes after row *N*).
2. If an occurrence of a character is picked from a row, you cannot pick the same occurrence again from that row.

QUICK EXPLANATION:

The main idea of this question is to traverse cyclic the matrix and do some string manipulation to know whether the string can be generated or not.

DETAILED STEPS:

1. Read the input as mentioned in input format.
2. Now, traverse the string to be generated.
3. For first char of string check whether it is present in the first row of matrix.
4. If present then remove that char from the first row of matrix and move row to the next row (Note: if row > n then use row % n).
5. If not present break the traversing of the string and print No.
6. Repeat steps 4 and 5 till the end of string is list

PROBLEM:

You will be given a string s.

You have to find the longest palindrome which can be formed by elimination of consecutive letters from between. i.e. you need to delete characters from index i to j where $j \geq i$, $i \geq 1$, $j \leq n$ (1 based indexing).

Print the longest palindrome possible after deleting a contiguous subsequence of $length \geq 0$ from between.

Rules:

1. You cannot add any new characters.
2. You can not change order of the original string.
3. Absolute difference of **number of characters chosen from beginning** and **number of characters chosen from end** of the string in the palindrome should be ≤1≤1.

Note: If more than one answer is possible, print lexicographically smallest one.

QUICK EXPLANATION:

We keep two pointers - $i = 1$ and $j = n$. We are checking the first and last character of the string and if they are equal we are incrementing the counter from left by one and decrementing the counter from right by one.

This is similar to trimming the extreme characters if they are equal.

If the counter from the left overtakes the counter from the right. i.e. they are adjacent to each other. This means that the string itself is palindrome.

The else condition is when we are no more getting similar characters from both ends and the counters have not reached next to each other.

$s[: i]$ represents the string till the counter i has travelled.

$s[j + 1 :]$ represents the string till which counter j has travelled. This string will be palindrome of the string $s[: i]$.

The middle part i.e. $min(s[i], s[j])$ gets the characters which are next to the strings which are included in the counter.

Now since the string is a palindrome, it will be of an even length. So we will take lexicographically smaller character between the two (1: which is in front of string $s[: i]$ and 2: which is before the string $s[j + 1 :]$)

Finally we print the answer.

PROBLEM:

In the given question chef wants to find if the order of the initials in his list and the order in which the employees are sitting can be matched by rotating the table or not. The table can only be rotated in a clockwise direction. Help the chef to find whether the list is correct or not.

QUICK EXPLANATION:

We can concatenate the given string with itself and check whether the string i.e the order after rotating the table is substring of the concatenated string or not

EXPLANATION:

Firstly we will create a temp string and store the concatenation of the first string.
Now we will check whether the string i.e the order of initials of employees after the rotation of the table is a substring of temp.
If the order of the employees after rotation of the table is the same as mentioned in the chef’s list then the string we will be the substring of the temp and the chef’s list is correct.
If it is not the substring of the temp then the chef’s list is wrong.

PREREQUISITES :

Array
String
Prime Number Theorem

PROBLEM :

Chef is a secret agent for the Government of Chef Land. Chef has found a gang planning to attack Chef Land. The gang is going to bombard few cities in Chef Land. **The names of the cities in Chef Land are always 7 digits numbers** .

The gang leader, Almoodi, is fond of mathematics and has devised a unique and perfect method of selecting the cities to be bombarded. **He has concatenated all prime numbers to create a big string: "23571113171923293137…".** He asks each of his bombers to provide a number (**which is not provided by previous bombers, to prevent collision**). This number is that starting index in the string of primes, and the city name will be the next 7 digits in the string. E.g., if the bomber chooses 8, the city name will be 1719232. Note that the index of string starts from 0.

Chef has a spy in the gang. The spy has provided Chef all the numbers chosen by the bombers. However, Chef isn't that comfortable with prime numbers and doesn't have time to learn it. So, Chef has requested you to assist him in finding the city names before it is too late.

SHORT EXPLANATION OF THE QUESTION :

Here basically all prime numbers are concatenated in a string.

- At first, you have to give input the number of the test cases (lets the input be n)
- Thereafter we have to give n number of inputs which are basically different indices (i) of the string.
- Now after getting each index we get the substring of the 7 characters from the i-th index (subString[i,i+7]).

QUICK APPROACH :

1. Take Input the number of the test case (t) and corresponding indices (n).
2. Now make a string that contains **n** th index and **subSting[n,n+7]**.
3. Then print the corresponding substrings of different indices.

EXPLANATION :

- At first let's take a **max** number 33000000 though 32000000 is sufficient for this problem
- Then make an array needed for sieve (**primeArr**) of bool type of size **max** which contains **0** (false) or **1** (true) if its ith index is non-prime or prime simultaniously.
- Then create the desired string of prime numbers by using the **primeArr**.
- Then take input the number of test cases (**t**) and maximum possible input (**n**) for each test case.

Time Complexity :

$O(m * loglogm)$

Space Complexity :

$O(m)$

Here **m** = $O(nlogn)$ which is the size of the array **primeArr**, array needed for sieve.

Now question is, how to find the value of m?

- **Let's Find the value of m :**
  We know (approximately) from the prime number theorem that

1. Numbers of prime numbers till n, π **(n) = n / log n**
2. n-th prime number p(n) = $n * (logn + loglogn)$
   But we don't need the nth prime number. Each prime number has some length. Most prime numbers will have length 6 for the given constraint for $n(1.5 * 10^7)$. We can cross-check this fact by putting n = 4, 5, and 6 in Eqn(3) for primes of length 4, 5, and 6 respectively.
3. Numbers of primes each having length x, Y(x) = $\pi(10^(x + 1)) - \pi(10^x)$
   Hence, we can divide p(n) by 6 to get an approximation of m.
   We have got approx value of m = $n * (logn + loglogn)/6$. For n = 10^7, m is approximately $4.8 * 10^7$.

We now try different values of m and print the length of the string of primes. We see that for m = $3.2 * 10 * 7$, length of string > n while for m = $3.1 * 10^7$, **length of string < n**. So, we shall use m = $3.2 * 10^7$.

PROBLEM:

A short and concise description of the problem statement.

QUICK EXPLANATION:

You are given a set of n pens, each of them can be red, blue, green, orange, and violet in color. You are given a string that contains ‘R’, ‘B’, ‘G’, ‘O’, and ‘V’ characters that represent the color of pens. You have to count a minimum number of pens that should be taken out from the set so that any two neighboring pens have different colors.

EXPLANATION:

The characters represent the color of pens.

We can get our desired sequence when neighboring pens have a different color. That is, No two neighboring pens should have the same color.

Thus, we will have to remove neighboring pens that have the same color to obtain our desired sequence of pens. For eg. let’s consider the sequence ‘RGRRBBVO’, in this sequence if we remove R at the 4th position, B at the 6th position. Then the resultant sequence will be ‘RGRBVO’ which is our desired sequence.

So, we will count the number of neighboring pens that share a common color.

For this, we will traverse the string and see if the current character is same as next character i.e
‘i’ th character is the same as ‘i+1’ th character, if both are the same then we will increase the count and after the traversal of string, we will print the count.

Let’s consider the same example ‘RGRRBBVO’, here R at the 3rd position is neighboring to R at 4th position. Similarly, B at the 5th position is neighboring to B at the 6th position. Thus we can see that two ‘i’ th characters are the same as ‘i+1’ th characters.

PROBLEM:

You will be given a bunch of numbers. You have to sort them according to Willow's rule.Now what is Willow's rule?

Basically he wants to sort the numbers in such a way that-

A is bigger than B

1)If the last digit of A is bigger than the last digit of B

2)If the last digit of A and the last digit of B are equal then he checks the previous one.

And thus the comparison continues.

Your task is to sort all the given numbers according to Willow's rule.

EXPLANATION:

Suppose there are two numbers A=52**1** and B=51**9**

From Willow's sort 9>1 so 519>521

It is as simple like this-We can reverse A and B thus we have

A=125 and B=915

Now if we compare them it is easy to figure out that A>B

So our task is to reverse all the given numbers and then sort them

And after sorting them we will again reverse them and will print them.

See the image below to figure out how the comparison works for 4500 and 45000 and why 4500>45000

TIME COMPLEXITY:

Highest number digit/string size , n=101(in worst case)
How many numbers/strings to compare , N=10^4(in worst case)
So.time complexity-

O(N*n)

PROBLEM:

You are given a string of length n consisting only of characters ‘a’, ‘b’ or ‘c’ where 1 <= n <= 10^5 . You have to find the maximum value of the absolute difference between i and j i.e abs(i-j) where s[i] != s[j] and 1<=i,j<=10^5.

QUICK EXPLANATION:

The maximum difference between indexes will be either between the rightmost character different from the first character or the difference between the leftmost character different from the last character.

EXPLANATION:

It was a basic observation-based question where we needed a greedy approach. It is clear that the distance between the rightmost character different from the first character or the distance between the leftmost character different from the last character will be the maximum distance.
We have to traverse the array twice and find both the distances and print the maximum between the two.
Time complexity will be O(n) where n is the length of the string.

PROBLEM:

Given a string S of n uppercase Latin letters, your task is to decide if it is possible to rearrange letters in S in such a way that the resulting string will have a string **LTIME** as a prefix and a string **EMITL** as a suffix.

QUICK EXPLANATION:

It is sufficient to check if S contains enough letters to form the required strings as its prefix and its suffix.

EXPLANATION:

In both subtasks, you have to handle some number of test cases.

SUBTASK 1

You have to handle up to 100 test cases, but the good news is that each one contains a string S of length at most 9. In this case, we can try any possible arrangement of letters of S, in other words, every permutation of its letters, and for each one, check if it has the required strings as a prefix and as a suffix. The total time complexity of this method is $O(T \cdot n! \cdot n)$ and it is enough, because n is at most 9 and we have at most 100 test cases to handle.

SUBTASK 2

The above method is only sufficient when dealing with such small inputs like we have in the first subtask. For the second subtask, it is way too slow, not only because we have up to 1000 test cases to handle, but mainly because n can be much larger here, up to 100. It is clear that we need a completely different approach here.

Let's think for a second, do we need to consider any permutation of letters of S? Of course we don't to do that. Notice that if there not enough letter to form both the required prefix and the required suffix, the answer is clearly **NO**. On the other hand, if there are enough letters in S to form the required prefix and the required suffix, the answer is **YES**.

First, we can notice that the shortest string containing required prefix and required suffix has length 9. This is true, because a string **LTIMEMITL** has length 9 and has the required prefix and suffix. Moreover, there is no such shorter string, because the longest prefix of **EMITL** which is also a suffix of **LTIME** has length 1.

Based on this observation, there are three cases to consider. If $n = 9$ then the answer is YES if and only if **S** contains two L, two T, two I, two M and one E. If $n > 9$, then the answer is YES if and only if **S** contains at least two L, at least two T, at least two I, at least two M and at least two E. If $n < 9$, then the answer is clearly NO.

The total complexity of this solution is $O(n)$, because the only thing we have to do is to count occurrences of 5 different letters in S.

PREREQUISITE:

Bitmasks

PROBLEM:

The notorious hacker group “Sed” managed to obtain a string S from their secret sources. The string contains only lowercase English letters along with the character ‘?’.

A substring of S is a contiguous subsequence of that string. For example, the string “chef” is a substring of the string “codechef”, but the string “codeh” is not a substring of “codechef”.

A substring of S is *good* if it is possible to choose a lowercase English letter C such that the following process succeeds:

- Create a string R, which is a copy of the substring, but with each ‘?’ replaced by the letter c. Note that all occurrences of ‘?’ must be replaced by the same letter.
- For each lowercase English letter:
  - Compute the number of times it occurs in S and the number of times it occurs in R; let’s denote them by A and B respectively. The ‘?’ characters in the original string S are ignored when computing A.
  - Check the parity of A and B. If they do not have the same parity, i.e. one of them is even while the other is odd, then this process fails.
- If the parities of the number of occurrences in S and R are the same for each letter, the process succeeds.

For different substrings, we may choose different values of C.

Help Sed find the number of good substrings in the string S.

Constraints

- $1 \le T \le 10^5$
- $1 \le S \le 10^5$
- S contains only lowercase English letters (‘a’ through ‘z’) and ‘?’
- the sum of $|S|$ over all test cases does not exceed 10^5

BRIEF EXPLANATION:

sm0l s0lution

Essentially, we check for all substrings. We keep track of the prefix parity using a bitmask. For every right index, we essentially want to count possible left indices. That can be done by just getting the number of prefix parity bitmasks that, when xorred with the current prefix bitmask, gives the bitmask of the parity of characters of entire string.

EXPLANATION:

Observation 1

First, we can represent all the parity information using a simple bitmask. One for each character, and one extra for parity of question marks.

Observation 2

Bitmasks also give us another advantage — we can xor them, which is commutative. Hence, we can keep track of prefix parities.

Observation 3 - Final Solution

Let the prefix bitmasks be represented by $bitmask[i]$ meaning parity of all characters in $s[1 \ldots i]$.

Checking for even occurances of ?

For checking substring with even number of $?$, we just check how many occurances of the same $bitmask[i]$ have occurred before …. but not exactly. What should we do?

Details

We should actually query with $bitmask[i] \bigoplus bitmask[n]$ since we want our bitmask to match that of the entire string.

Checking for odd occurances of ?

Here, try to replace which each character (think how the bitmask to query changes, also remember to flip the bit representing the ?).

Intuition on why the methods are different

See Codes for exact implementation details.

PROBLEM

There is a dollar note array whose values need to be distributed between Mr. White and Pinkman. If the toss result is H, Mr. White picks a dollar note and then Pinkman. Alternatively, if the toss results in T, Pinkman selects first and then Mr. White. We need to find the person with the maximum amount of money at the end of the sharing procedure and the share he gets.

QUICK EXPLANATION

The dollar note array can be sorted in descending order so that at any point Mr. White or Pinkman make the optimal selection. We have to add the total money accumulated by Mr. White and Mr. Pinkman as the dollar notes get divided between the two. At the end, whoever gets the greater share of money, wins.

EXPLANATION

We have to find the amount of money Mr. White and Mr. Pinkman have accumulated after the sharing procedure. The **optimal choice for both of them is to pick the maximum value remaining in the dollar note array**. So, for the ease of optimal choice selection, we can sort the dollar note array in descending order.

Now, **if toss result is H: Mr. White picks first** and the first value in the array (which is the optimal one since we have sorted it in the descending order) gets added under his belt. Then Mr. Pinkman gets to choose and he will get the next highest value.

If toss result is T: Pinkman selects first and the highest value in the array gets transferred to him. Then Mr. White selects and takes the next highest dollar.

Once you have traversed the entire dollar note array, print the name of the person whose share of money is greater. And if both the values are same (in case of a draw), then print -1.

If N ($no.\ of\ dollar\ notes = size\ of\ dollar\ note\ array$) is odd, then we need to keep extra checking till the last toss. To avoid it, **we can always add an extra dollar to our notes array with value 0**. So, ultimately whoever gets this, his share remains unchanged (which is equivalent to not getting anything).

TIME COMPLEXITY

O(n)

PROBLEM:

A string is given in the form of ‘H’, ’T’ and ‘.’ characters, where H denotes the head, T denotes the tail and ‘.’ denotes the time between a snake passing or a the time between one snake and another. Given a string, check if it is a valid sequence of ordering of snakes.

EXPLANATION:

Let’s see… the ‘.’ characters don’t look useful. Do they? Nah. They’re just signifying empty spaces… which doesn’t help us at all. Ignore them.

Now, we have a string made up entirely of ‘H’ and ’T’ characters. This string denotes the ordering of the snakes. Alright, so when is a string invalid?

So, like most animals, a snake can’t have two heads or tails. Thus, we can’t have two ‘H’ or ’T’ characters together. What else?

Let’s see, we can’t have a tail appearing before a head, that’s preposterous! Similarly, we can’t have a head without a tail. Could these two cases appear anywhere in our string?

Sadly, they can. The first case will appear only at the beginning of the string. The second will appear towards the end.

Thus, we can simply check if any of the above conditions is true, thus making our answer ‘Invalid’. Otherwise, our answer is ‘Valid’. We have a cute solution.

Simply put, the only way a string is valid is when it is of the form “HTHTHTHT”. Starts with an ‘H’, ends with a ’T’ and no character should repeat right after itself.
PS: Finding edge cases is always important. Take whatever is in a problem statement at face-value and don’t assume that a case isn’t possible. To put it in Sherlock’s words - “If you eliminate the improbable, whatever remains must be the truth”. Always be on the lookout for cases where your code could fail.

SOLUTIONS:

Editorialist’s solution - // SNAKPROC// Aulene De#include<iostream>#include<fstream>#include<cst - Pastebin.com 105

PROBLEM:

We are given the Burrows-Wheeler Transform (BWT)[Burrows–Wheeler transform - Wikipedia 11] of a binary string, and we have to reconstruct the first row of the sorted array.

BACKGROUND:

The technique of sorting all the circular shifts of a text in a lexicographic order and then extracting the last column, describes the Burrows-Wheeler Transform of a string, which is usually used to make a string easy to compress. The following article however presumes no previous knowledge of Burrows-Wheeler Transform.

EXPLANATION:

To understand the solution to this question, we will start by re-doing the given example in the question, with a slight modification. Instead of considering it as a binary string, lets for consider for a moment that each character is a unique letter. We do so by putting a subscript on each of the character. The subscript ***i*** on a character corresponds to the ith occurrence of the character in the string (i.e. the rank of the character!)

So now we have to transform the string $\mathbf{0_1\ 0_2\ 1_1\ 1_2\ 0_3}$

Initial Array	Sorted Array	Rightmost column
$0_1\ 0_2\ 1_1\ 1_2\ 0_3$	$0_3\ 0_1\ 0_2\ 1_1\ 1_2$	1_2
$0_2\ 1_1\ 1_2\ 0_3\ 0_1$	$0_1\ 0_2\ 1_1\ 1_2\ 0_3$	0_3
$1_1\ 1_2\ 0_3\ 0_1\ 0_2$	$0_2\ 1_1\ 1_2\ 0_3\ 0_1$	0_1
$1_2\ 0_3\ 0_1\ 0_2\ 1_1$	$1_2\ 0_3\ 0_1\ 0_2\ 1_1$	1_1
$0_3\ 0_1\ 0_2\ 1_1\ 1_2$	$1_1\ 1_2\ 0_3\ 0_1\ 0_2$	0_2

Table 1: BWT of modified string

The transformed string is: $\mathbf{1_2\ 0_3\ 0_1\ 1_1\ 0_2}$

We'll make some crucial observations now,

1. Since we rotate the string by one place in every row, the number of rows is same as the length of the given strings and the each of the unique characters in the initial string occur only once in the final string formed out of the rightmost column. Hence, from the transformed string, we can deduce the frequency of 0's and 1's in the original string!
2. In the sorted array, the rows were sorted lexicographically, this results in all the 0's followed by all the 1's in the first column of the sorted array. Hence, we can deduce the leftmost column of the sorted array from the given rightmost row!
3. On closely observing the first and the last column in the sorted array, we find that the relative ordering of 0's is same in both of them ($0_3\ 0_1\ 0_2$). Same goes for the 1's.

Sorted Array

0_3 0_1 0_2 1_1 1_2

0_1 0_2 1_1 1_2 0_3

0_2 1_1 1_2 0_3 0_1

1_2 0_3 0_1 0_2 1_1

1_1 1_2 0_3 0_1 0_2

Table 2: Sorted array from table 1

That being said, we can rank the characters of the initial string in such a manner, that the ranks occur in a sorted order.

First Column	Last Column
0_1	1_1
0_2	0_1
0_3	0_2
1_1	1_2
1_2	0_3

Table 3: First Column and Last Column of sorted array with modified ranks

Constructing the rest of the string:

We will construct the rest of the string from the knowledge of the first and last column.

Since, the rows are formed by rotating the string, the character in the last column is the character which must occur before character in the first column of the corresponding row.

So 1_1 occurs before 0_1

0_1 occurs before 0_2

0_2 occurs before 0_3

0_3 occurs before 1_2

1_2 occurs before 1_1

Since the topmost row starts with a 0_1 and ends in 1_1(Table 3-first row).

Using the cycle above, we obtain, the topmost row: 0_1 0_2 0_3 1_2 1_1

Answer: 0 0 0 1 1.

Subtask 1:

S = 1 0 1 1 1 1 0 0

First Column	Last Column
0_1	1_1
0_2	0_1
0_3	1_2
1_1	1_3
1_2	1_4
1_3	1_5
1_4	0_2
1_5	0_3

Answer: 0 0 1 1 0 1 1 1

Subtask 2:

S = 1 0 1 1 0 1 1 1 0 0 1 0

First Column	Last Column
0_1	1_1
0_2	0_1
0_3	1_2
0_4	1_3
0_5	0_2
1_1	1_4
1_2	1_5
1_3	1_6
1_4	0_3
1_5	0_4

First Column	Last Column
1_6	1_7
1_7	0_5

Answer: 0 0 0 1 1 1 0 1 1 0 1 1

Subtask 3:

S= 1 1 1 1 0 1 1 0 1 0 1 1 0 1 0

First Column	Last Column
0_1	1_1
0_2	1_2
0_3	1_3
0_4	1_4
0_5	0_1
1_1	1_5
1_2	1_6
1_3	0_2
1_4	1_7
1_5	0_3
1_6	1_8
1_7	1_9
1_8	0_4
1_9	1_10
1_10	0_5

Answer: 0 0 1 1 1 1 0 1 1 1 0 1 0 1 1

Reply

Created	Last Reply	Replies	Views	Users	Likes	Links
Jan '20	Dec '20	3	1.1k	2	3	3

11 months later

b23v Dec '20

mazumder7:

That being said, we can rank the characters of the initial string in such a manner, that the ranks occur in a sorted order.

First Column	Last Column
01	11
02	01
03	02
11	12
12	03

How did you make this table ? I dont understand what first and last column means in this table, how did you rank ? Can you clarify ?

mazumder7 Dec '20

In this table, the last column refers to the last column of the BWT matrix (the binary string provided as an input to you in the question); the first column similarly, refers to the first column of the BWT matrix. In the question, we have been provided with the last column. Using the last column we can very easily deduce the first column using the 3 crucial observations we made above!

Reply

b23v Dec '20

mazumder7:

Reply

Initial Array	Sorted Array	Rightmost column
01 02 11 12 03	03 01 02 11 12	12
02 11 12 03 01	01 02 11 12 03	03
11 12 03 01 02	02 11 12 03 01	01
12 03 01 02 11	12 03 01 02 11	11
03 01 02 11 12	11 12 03 01 02	02

Table 1: BWT of modified string

Maybe I am missing something, but the first column of the sorted matrix is different from what you have written in the table.

PREREQUISITES:

String, Math

PROBLEM:

Given a line of string, You have to define the roll number of the student who shouted with that string. Suppose the student with roll no. 1 will say **aoreee gelooo** similarly **zaoreee gelooo** will be said by roll no. 26 and next student of him will say **aaoreee gelooo**. 52nd student will say **azoreee gelooo** and next student will say **baoreee gelooo**.

QUICK EXPLANATION:

There are two string, second one is absolutely unnecessary. Last 5 character of the first string is also unnecessay. Take the remaining characters. Then take the value of the characters in lexicographical order as a=1, b=2, c=3. Suppose for **abcoreee gelooo**, take abc and it is 123 and it is a 26 based number. Convert it to 10 based number and it is the answer.

EXPLANATION:

Students will remove first character from the first word of **"Poreee gelooo"** and replace with one or more character into it. **"gelooo"** have no use here. So take the word **"poreee"** in a string and reverse it (example: poreee will be reversed to eeerop). Now skip first 5 character **"eeero"** and start from the 6th. Just type cast each characters to its integer value as a=1, b=2,… and multiply the value with the power of 26. For the first character(6th in the reversed word), you should multiply with 26^0 and for the second character(7th in the reversed word), it will be 26^1 and so on.

Why power of 26? Think of a decimal number, suppose it is 237. How it is represented with its positional value? Simply 2x(10^2)+ 3*(10^1) + 7*(10^0) = 237. 237 is a 10-based number so we used power of 10. But there are 26 character from a to z. So we used the power of 26.

Let's clear with an example, for abcoreee gelooo. reverse abcoreee and you will find eeerocba. Now skip eeero and get cba. Now c=3, b=2, a=1. So it will be, 3*26^0+ 226^1 + 1*26^2 is the answer.

Time complexity: O(n)

PROBLEM:

One day Rashi went to the kitchen. She took out N strings from the cooker and kept the empty cooker on the stove. But Rashi is clever and she has put the entire blame for this on Gopi. To save herself from the false accusations of Rashi, Gopi must answer M queries asked to her. In each of the M queries, she is given an integer K and she needs to tell the number of possible strings which are prefix of at least K strings out of the given N strings.

EXPLANATION:

Subtask-1:

The implementation for the first subtask is quite clear. Just iterate over all the (n*length of string^2) prefixes and count their no of occurrences in all the strings. Then build a suffix array of length n for all the values of k given in the question and return array[k] as the answer of each query.

Subtask-1 &2

Due to the tight constraints the iteration over all the (n*length of string^2) possible prefixes would not fit into the time limit ,we need to optimise the solution using a prefix trie.

We will build a prefix trie and insert all the n strings in the trie and then we would traverse the trie and store the answer for each k in an array by incrementing the value of arr[word count] at each node of the trie and then we would convert this array in a suffix array and return the arr[k] as the answer for each query.

Time Complexity-O(n*(length of string)+m)

Space complexity-O(n*(length of String))

PREREQUISITE
Compression, Dijkstra, Strings, DP, Shortest Path Algorithm etc.

PROBLEM
Given an array of positive integers representing 2-D bar heights, design an algorithm (or write a function) that can compute the total volume (capacity) of water that could be held in all lakes on such an island given an array of the heights of the bars. Assume an elevation map where the width of each bar is 1.

EXPLANATION

Using the Dijkstra algorithm, which finds the shortest paths between nodes in a graph

Consider a function `f(x)` which for a `node x` tells you what the water level is for that node (if you subtract the height `h(x)` at that point and integrate over the whole are then you have the answer)

Now `f(x) = min_{path p from x to answer sinkhole} max_{q in p} h(q)`

So this is a version of the Dijkstra algorithm using the edge weight max instead of the usual sum.

O(N) Solution: Scan left to right, then right to left

Overall strategy: We make use of the fact that every graph consists of an uphill section, and a downhill section. A single pass will only work on an “uphill” section of the graph. So we solve the left uphill portion with a left to right scan, and then reverse direction to solve the right downhill section with a right to left pass, thus turning the downhill section into an uphill section.

First pass: Scan left to right keeping track of maximum possible water level. Each time a bar greater than or equal to the height of the water level is found, accumulate the lake so far, and update the water level. If no such bar >= the waterlevel is found, do not accumulate the lake. The final bar will always constitute such a condition. The first pass finds the volume of the lakes to the left of the rightmost bar of max height. Record the position of the final max height bar (to the right of which there exists no bar of greater or equal height).

Second pass: We now iterate from right to left (keep the running total from before). We already know the total volume of the lakes identified in the first pass. On the second pass, we find the volume of the remaining lakes.

The second pass terminates when we reach the final max height bar from the 1st pass. Take care to accumulate the final lake.

This approach breaks the tracking of “interesting” columns into a separate step. Doing that costs more time, but might well make for more-readable code.

Walk in from the left edge and the right edge simultaneously.

This approach relies on always knowing that the leftmost (rightmost) edge we’re looking at *must* be on the low side of any basin. Then alternatively work either the left side or the right side towards the middle depending on which one is lower, so as to preserve that invariant.

When left meets or crosses right, we’re done.

SOLUTION
The Solution for this problem can be found here [1]

Zonal Informatics Olympiad, 2011

Instructions to candidates

1. The duration of the examination is 3 hours.
2. Calculators, log tables and other aids are not permitted.
3. The question paper carries 80 marks, broken up into four questions of 20 marks each. Each question has three parts. *If you solve all three parts correctly, you get 20 marks for that question.* Otherwise, you get 5 marks for each part that you solve correctly.
4. Attempt all questions. There are no optional questions.
5. There is a separate *Answer Sheet*. To get full credit, you *must* write the final answer in the space provided on the Answer Sheet.
6. Write *only* your final answers on the Answer Sheet. Do *not* use the Answer Sheet for rough work. Submit all rough work on separate sheets.
7. Make sure you fill out your contact details on the Answer Sheet as completely and accurately as possible. We will use this information to contact you in case you qualify for the second round.

Zonal Informatics Olympiad, 2011

Questions

1. A new IT Park has N north-south roads and N east-west roads laid out in an $N \times N$ grid. A security guard who is posted at the intersection of two roads can observe all activity along the length of both roads.

   There are N guards. Initially, the guards are placed haphazardly at some of the intersections. *The guards have to be moved into new positions so that each road is under observation by at least one guard.* While moving the guards around, we also have to ensure that we never have two guards at the same intersection.

   The aim is to do this in such a way as to minimize the total distance travelled by the guards. We count the distance travelled by guards in *steps*: in one step, a guard can shift to an adjacent intersection.

   For example, suppose there are 4 roads in each direction and 4 guards placed as shown in Figure 1. Here, for instance, the rightmost north-south road and the bottom east-west road are not observed by any of the guards.

   We can move the guard at (3,2) down and across to (4,4) in three steps to get a solution, as shown in Figure 2.

   Another possibility is to move the guard at (3,3) to (3,4) and then move the guard at (3,2) to (4,3). This also takes three steps.

Figure 1

Figure 2

Figure 3

In each of the following cases, you are given the intial positions of the N guards on an $N \times N$ grid of roads. You have to compute the minimum number of steps required to move the guards into new positions so that all roads are under observation. In the example above, it can be shown that the minimum number of steps required is 3.

(a)

(b)

(c)

2. Given a number N between 2 and 9, consider the integers that can be formed using each digit in the set $\{1, 2, \ldots, N\}$ exactly once. For instance, if N is 3, the integers we can form are 123, 231, 312, 132, 321 and 213. If we arrange these in increasing order, we get the list $\{123, 132, 213, 231, 312, 321\}$. The fourth number in this list is 231.

   In general, given two numbers N and K, the task is to compute the number at position K when all integers formed using the digits $\{1, 2, \ldots, N\}$ exactly once are arranged in ascending order. For instance, the example worked out above corresponds to $N = 3$ and $K = 4$.

Compute the answer for the following values of N and K.

(a) $N = 5$, $K = 76$ (b) $N = 7$, $K = 4197$ (c) $N = 9$, $K = 191082$

3. A cable TV operator is trying to save money by running cables directly at roof level between pairs of buildings on the same side of the street. A cable can be connected between the rooftops of two buildings A and B if no building between them is strictly taller than either A or B.

   For instance, suppose there are five buildings on the street with the following heights (in feet): 160, 145, 153, 170, 180. Let us label the five buildings A, B, C, D and E. In this case, six pairs of buildings can be connected by a TV cable at roof level: (A, B), (C, D), (A, C), (B, C), (D, E), (A, D).

   In each of the following cases, you are given the heights of the buildings on the street and you have to compute the *number of pairs* of buildings that can be directly connected by a TV cable at roof level. In the example above, the answer is 6.

   (a) 168, 92, 120, 147, 73, 160, 156, 108, 145, 157, 71, 71, 109, 157, 152, 214, 191, 78, 154, 186

   (b) 230, 142, 176, 225, 111, 163, 175, 241, 72, 76, 99, 145, 146, 82, 153, 118, 158, 239, 86, 246, 156, 98, 154, 83, 205

   (c) 196, 98, 134, 169, 73, 185, 289, 168, 262, 291, 72, 71, 120, 181, 122, 162, 147, 75, 124, 144, 106, 100, 224, 139, 229, 134, 87, 156, 251, 150

4. Consider a sequence $b_1 b_2 \ldots b_N$ of N binary digits. Given such a sequence, we rotate it by one digit to the left $N-1$ times to generate a block of N sequences arranged in an $N \times N$ array of 0's and 1's, as follows.

$$\begin{array}{lllll} b_1 & b_2 & \ldots & b_{N-1} & b_N \\ b_2 & b_3 & \ldots & b_N & b_1 \\ \vdots & \vdots & & \vdots & \vdots \\ b_{N-1} & b_N & \ldots & b_{N-3} & b_{N-2} \\ b_N & b_1 & \ldots & b_{N-2} & b_{N-1} \end{array}$$

   We then sort these sequences in lexicographic order—that is, regard each row of the array as a binary number and rearrange the rows in ascending order.

   After this rearrangement, we extract the last column of the new $N \times N$ array. The goal is to work backwards from this column and compute the top row of the $N \times N$ sorted array that created it.

   For example, consider the binary sequence 0 0 1 1 0. After sorting the rows of the 5×5 array that this sequence generates, the last column reads 1 0 0 1 0 from top to

bottom. Given this last column, the task is to determine the first row of the sorted array, which is 0 0 0 1 1. The example is illustrated below.

Initial array	*Sorted array*	*Rightmost column*
0 0 1 1 0	0 0 0 1 1	1
0 1 1 0 0	0 0 1 1 0	0
1 1 0 0 0	0 1 1 0 0	0
1 0 0 0 1	1 0 0 0 1	1
0 0 0 1 1	1 1 0 0 0	0

Each of the inputs below describes the rightmost column of a sorted array, from top bottom. Your task is to compute the first row of the corresponding sorted array for each of these inputs. In all three cases, the answer is unique.

(a) 1 0 1 1 1 1 0 0

(b) 1 0 1 1 0 1 1 1 0 0 1 0

(c) 1 1 1 1 0 1 1 0 1 0 1 1 0 1 0

Zonal Informatics Olympiad, 2011: *Answer sheet*

Name:	**Class:**	**Sex:**
School:		
Examination Centre:		
Father or Mother's Name:		
Full home address with PIN code:		
Home phone number, with STD Code:		
Email address:		

Write only your final answers in the space provided. Write all rough work on separate sheets.

1. (a) Minimum number of steps: ☐ (b) Minimum number of steps: ☐

   (c) Minimum number of steps: ☐

2. (a) $N = 5$, $K = 76$ ☐

   (b) $N = 7$, $K = 4197$ ☐

   (c) $N = 9$, $K = 191082$ ☐

3. (a) Number of pairs: ☐ (b) Number of pairs: ☐

   (c) Number of pairs: ☐

4. (a) First row ☐

   (b) First row ☐

   (c) First row ☐

For official use only. Do not write below this line.

1.	a	b	c		2.	a	b	c	
3.	a	b	c		4.	a	b	c	

Total