Problem Tutorial: "Homework"

Denote the number of characters in A as n. If some character occurs more than $\frac{n}{2}$ times, then the answer is "`IMPOSSIBLE`", because, in this case, the sets of positions of this character in the first string and in the second string will necessarily intersect.

Otherwise, the answer exists. There are many different constructions; we'll describe the simplest known to us. You can choose any B, but it's the most convenient to take the sorted version of A. This implies that the set of occurrences of every character is a contiguous segment. After that, take C to be equal to B cyclically shifted by $\frac{n}{2}$. This way, the segment for every character also shifts by $\frac{n}{2}$, but, since the number of occurrences of every character is at most $\frac{n}{2}$, these segments in B and C don't intersect.

Here is an implementation of this solution in Python:

```
A = input()
n = len(A)
B = sorted(A) # Sort A
C = B[n//2:] + B[:n//2] # Set C to be a cyclic shift of B by n // 2
if any(B[i] == C[i] for i in range(n)): # Check if the answer is correct
    print("IMPOSSIBLE") # If no, print IMPOSSIBLE
else:
    print("".join(B)) # If yes, print B and C
    print("".join(C))
```

Problem Tutorial: "Matryoshka Inc"

Let's reformulate the problem: we are given an array of numbers (possibly with leading zeroes), and we can arbitrarily permute digits in each number. The task is to maximize the length of the longest increasing subsequence (LIS).

Finding LIS is a well-known problem solved by dynamic programming. Still, the classic solution doesn't work here: we can't only store `dp[i]` because we're also interested in the actual value of the last element in the subsequence. However, if we include the value in the DP state, we can pass some subgroups.

The solution here also uses dynamic programming, but differently: let's iterate over all integers from left to right (let the current index be i). We'll also maintain array `dp[j]` — the minimum last integer in the increasing subsequence of length j. This is enough to make the transition to the next integer.

How do we process another integer from the array? Let's try all possible j, the length of the increasing subsequence that we would like to append our new element. Now we need to permute digits in the $i+1$-th integer so that we have an integer larger than `dp[j]` and the minimal possible among those. This is also a pretty standard problem: we try to match the longest possible prefix, and, on the next position, we put a digit that's larger (but still as small as possible). Minimally fill the rest. We'll call this function for all j.

The time complexity is $O(n^2B)$, where B is the maximum decimal length of input integers.

Problem Tutorial: "Password Lock"

Let's look at their remainders instead of numbers on the cells when divided by k. We can almost always put numbers with the same remainder side by side, except for the cases of remainder 0 and $\frac{k}{2}$ (if k is even).

Let's first try to put all the cells in ascending order of the remainder. Then we can have three places that do not fit the condition:

1. two remainders 0 side by side;
2. two remainders $\frac{k}{2}$ side by side (this can happen only if k is even);

3. two remainders x and $k-x$ side by side, x $\not k-x$ (this can be at most in one place).

Let's temporarily remove all cells with a remainder of 0 and $\frac{k}{2}$. To get rid of the third case, you need to insert any other remainder between these elements. You can take either 0 (if it exists), or $\frac{k}{2}$ (if it exists), or a cell from the beginning (if there is another remainder), or a cell from the end (if there is another remainder). We take precisely from the beginning or the end so as not to spoil anything and not to add a third case in other places. It is clear that if we have only two different remainders, a suitable sequence does not exist.

Now let's try to insert back cells with remainders 0 and $\frac{k}{2}$. Let there be fewer zeros (in the case of a minority of $\frac{k}{2}$, we act similarly). Then paste them all at the beginning. Then paste all $\frac{k}{2}$ through one. Since there are more of them, there will be no adjacent zeros. If $\frac{k}{2}$ is too much, and we can't insert everything, then this means that there are more than half of all cells; then a suitable sequence does not exist.

Thus you got rid of all three possible cases.

Problem Tutorial: "The Name of the Fourth Problem"

In mathematics, this sequence is known as *Golomb sequence*. To solve the first few subtasks, you need to figure out how this sequence works and turn it into code (or find the explicit recurrent formula $a(n) = 1 + a(n - a(a(n)))$); implement prefix sums.

To solve this problem for larger n, we need to use the specific structure of this sequence. Notice that it has a lot of repetitions that come in runs of adjacent elements. We can use `Run-length encoding`. Instead of storing each number explicitly, we will work with pairs (value, count). It's a bit harder to calculate prefix sums on these pairs, but still not impossible: we calculate prefix sums for counts (to find the bound with binary search) and total sums (in other words, for value times count products). This solution works for $r_i \le 10^{10}$.

To receive the full 100 points, we need to make one similar step. Let's look at segments in the previous solution. Using the same argument, the "count"s in these segments monotonically increase and have a lot of repeats. In fact, the sizes of segments from the previous solution are precisely the original sequence (because it is self-describing). So let's do the compression again, but this time, on segments. In the end, we'll store the sequence as blocks "numbers from a to b inclusive; each is repeated k times". Similarly, we can build prefix sums on these blocks and use binary search with prefix sums to answer the RSQ queries.

The time complexity consists of the precalculation (we need to calculate at most $L < 10^6$ blocks) + $O(\log L)$ per query.

Problem Tutorial: "Comparing Theories"

First of all, we will count similar triples instead. Then, in the end, we will subtract their count from $\binom{n}{3}$.

The solution that works in $O(n^4)$ or $O(n^3)$ might work as follows: try all triples of leaves, and compute the "middle" vertex for them. It will be a parent of some two out of three leaves; check that this pair coincides in both trees. The "middle" vertex is the one that appears exactly once in $(\mathrm{LCA}(A,B), \mathrm{LCA}(A,C), \mathrm{LCA}(B,C))$. Depending on the implementation of LCA, this solution's time complexity is between $O(n^4)$ and $O(n^3)$.

How to count these triples faster? Consider LCA of three leaves in the first tree. This vertex has two subtrees: color one subtree in red and another in blue. Everything outside this subtree will be colorless. Fix a vertex in the second tree; suppose it's also a LCA of three leaves in the second tree.

How many ways are there to choose three leaves, such that their structures coincide, and their LCAs in the first and the second tree are these two vertices?

For the LCA of three leaves, two out of three leaves must come from one subtree and one — from another. Let R_1, R_2, B_1, B_2 be the number of red/blue leaves in the left/right subtree in the second tree. Then,

the number of shared triples is $\binom{R_1}{2}B_2 + \binom{R_2}{2}B_1 + \binom{B_1}{2}R_2 + \binom{B_2}{2}R_1$. Summing this over all vertices of the second tree (in linear time) gives us a $O(n^2)$ solution.

Imagine that we have a data structure that allows us to perform two types of operations:

- Change a leaf's color.
- Calculate the formula above for all vertices and return the sum.

We'll discuss how to implement such a data structure later. First, let's discuss how to solve the problem in $O(n \log n)$ queries to this structure. Run the depth-first search that colors all leaves in red and blue for every vertex in the first tree. It will obey two conditions: before entering v, every leaf in its subtree is red, and everything else is colorless; after exiting v, every leaf is colorless. It works as follows:

1. Recolor all leaves in the **smaller** subtree v into blue, query the data structure for the answer for the current vertex.
2. Uncolor everything in the smaller subtree.
3. Run recursively for the **larger** subtree; its prerequisite is fulfilled.
4. After exiting the larger subtree, color all leaves in the **smaller** subtree v into red, and run recursively.
5. Both recursive calls cleared their respective subtree; it's safe to exit.

Inside each DFS call, we make $O(k)$ queries, where k is the size of the smaller subtree. It's well known that the sum of these is $O(n \log n)$ queries.

The data structure in question can be implemented using Heavy-Light decomposition ($\log^2 n$ or $\log n$ per query), where you need to change a leaf's color and update the count on the path to the root. Also, since this problem is significant in Computational Biology, plenty of papers discuss the algorithms for triplet/quartet distance. One of the data structures, called *Hierarchical Decomposition Tree*, can also answer this query in $O(\log n)$ time per query. You can read the details at `https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-14-S2-S18`.