Problem :

Joey likes to play with strings. He loves asking others for letters and making new words (*by shuffling the letters as he wishes*) out of them which he then adds to his *personal dictionary*. And as with everything involving Joey, his dictionary is special too of course. It only contains words that are **Palindrome** in nature (*If the string is not palindrome in nature, it is discarded and is not used in his dictionary*).

Now Ross seeing this, decides to take on a tougher challenge and decides to help Joey by using all the letters that are given to him and adding it to Joey's dictionary if its a palindrome & if its not a palindrome, to **add extra alphabets** (assigned as - -*modified*) to the string to make a word that's palindrome in nature.

Now, it being Ross , as it is with most of his cases, he fumbles over his task & so requires our help to provide new words for Joey by adding minimum number of alphabets & adding the alphabet in lexicographical order.

Difficulty : Easy

Prior Knowledge :

This requires a basic knowledge of **String manipulation** and how a **palindrome** is formed.

Approach :

Here, we utilise the common logic of how many letters of a certain alphabet should be there for a string to be palindrome. Here 2 cases arise :

Case 1 : For **EVEN** Length of string. In such a case, the count of all alphabets in the string must be even.

Case 2 : For **ODD** Length of string. In such a case, the count of all alphabets in the string must be even *except* for the character in *middle* of the palindrome which has an Odd character count.

So, using these rules, we ensure that any string given to us can be converted into a palindrome. The process to do this mainly comprise of 3 Steps :

Step 1 : We get a **count** of how many times each *Alphabet* occurs in the string by taking index **0** in an array (of size 26 as we only have lower case letters as input) as **char 'a'**, **1** as **char 'b'** and so on and so forth.

Step 2 : Keeping the last one alphabet that is Odd as it is, we convert any other alphabet that has Odd count to **Even** by **adding 1** to it.

The reason we keep one alphabet *Odd* is to maintain the smallest string possible and that happens to be odd length string in this case

Step 3 : **Print** the characters from Array 0 to 26 one by one depending upon the count each one has. This takes care of the Lexicological Ordering too.

<u>CODE</u> :

```
//------------------------------KAUSHIK_THE_DEVELOPER------------------------------


//#include<bits/stdc++.h>

#include<iostream>

#include<string>      //Obj

#include<vector>

#include<algorithm>

using namespace std;


int main()

{

   long long t,n,w,i,j,k,a[26],f=0,count;
```

```
string s,ch;
cin>>t;
for(w=0;w<t;w++)
{
   count=0;
   fflush(stdin);                    //<stdio.h>
   cin>>s;
   n=s.length();
           //Initalise the character count for all 26 Alphabets as 0
   for(i=0;i<26;i++)
       a[i]=0;
   //Find out count of each alphabet in string
   for(i=0;i<n;i++)
   {
       k=int(s[i])-int('a');
      a[k]++;
   }
   //Adjust no. of alphabets count - Going from 'z' to 'a'
   f=0;         //Flag - To leave one alphabet as odd
   for(i=25;i>=0;i--)
   {
      if(a[i]%2==1 && f==0)        //Odd no. count - Keep one unchanged
      {
               f=1;              //Flag set
              continue;
      }
      else if (a[i]%2==1)
      {
                  a[i]++;
                  count++;
      }
}
if(count>0)
   cout<<"MODIFIED ";
else
   cout<<"ORIGINAL ";
//Print the string in ascending order (Lexological order)
for(i=0;i<26;i++)
{
          if(a[i]>0)
      {
                   ch=char(i+int('a'));
                  for(j=a[i];j>0;j--)
                  {
                                 cout<<ch;
                   }
   }
}
if(count>0)
        cout<<" "<<count;
        else    cout<<" 0";

        cout<<endl;

    }

    return 0;

}
```