A

A nice way to think about the problems in this week's contest is to use the idea of **canonical forms**.

As an analogy, two fractions a/b and c/d are considered equivalent if $ad = bc$... but the more common way taught to elementary school children is to *reduce to lowest terms first*—two fractions are equivalent if, after reducing both to lowest terms, they are exactly equal.

For this problem, we (arbitrarily) decide that the canonical form of each word uses `e` instead of `i`, and `o` instead of `u`—let's call this the `eo` form of a word. Then, two words are equivalent if and only if, after reducing both to their `eo` forms, the two words are exactly equal.

For example:

- `rule` and `roli` are equivalent because both become `role` when reduced to their `eo` form.
- `mabuti` and `mabote` are equivalent because both become `mabote` when reduced to their `eo` form
- `eua` and `uea` are **not** equivalent, because their `eo` forms of `eoa` and `oea` are not equal.

This lets us use the normal builtin string equality operator `==` of our programming language to check for equivalence.

```
# map i -> e and u -> o; everything else is untouched
def canon_letter(c):
    if c == 'e' or c == 'i':
        return 'e'
    elif c == 'o' or c == 'u':
        return 'o'
    else:
        return c

# apply the canon_letter transformation to each letter
def canon(w):
    return ''.join(canon_letter(c) for c in w)

s, t = input().split()
s = canon(s)
t = canon(t)
print(
    "yis"
    if s == t
    else "nuu"
)

```

B

Let's read the word and then reduce it to its `eo` form. How many different words reduce to this `eo` word? Well, look at how we implement that `canon_letter` function.

- The letter `e` could have come from an `e` or `i` (using jargon: the pre-image of `e` is `{e, i}`
- The letter `o` could have come from an `o` or `u` (using jargon: the pre-image of `o` is `{o, u}`
- Every other letter c remained untouched (using jargon: the pre-image of every other c is $\{c\}$).

Since each letter is transformed independently of the others, the answer is:

$$\text{choices for 1st letter} \times \text{choices for 2nd letter} \times \cdots \times \text{choices for last letter}$$

But given the structure we just discussed, this simplified neatly into:

$$2^{\text{number of } \texttt{e} \text{ or } \texttt{o} \text{ in } \texttt{eo} \text{ form}} \times 1^{\text{number of other letters}}.$$

Of course, $1^n = 1$ for all naturals n, so that factor can be ignored.

Note for C++ and Java users: Be careful of integer overflow! If the entire word consists of `e` and `o`, then the answer could be as high as 2^{60}, which **does not fit** in a 32-bit data type like `int`. You must use a 64-bit data type.

```python
# map i -> e and u -> o; everything else is untouched
def canon_letter(c):
    if c == 'e' or c == 'i':
        return 'e'
    elif c == 'o' or c == 'u':
        return 'o'
    else:
        return c

# apply the canon_letter transformation to each letter
def canon(w):
    return ''.join(canon_letter(c) for c in w)

s = canon(input())
print(2**sum(1 if c in 'eo' else 0 for c in s))
```

K

Again, first reduce all words to their canonical `eo` form. The problem is now much easier to think about—we just want to count the number of distinct words in our list, removing duplicates (since duplicates can just go on the same sheet as a pre-existing word).

You can remove duplicates using a builtin data structure like `set`.

```
# map i -> e and u -> o; everything else is untouched
def canon_letter(c):
    if c == 'e' or c == 'i':
        return 'e'
    elif c == 'o' or c == 'u':
        return 'o'
    else:
        return c

# apply the canon_letter transformation to each letter
def canon(w):
    return ''.join(canon_letter(c) for c in w)

n = int(input())
print(len(set(canon(input()) for _ in range(n))))

```

D

Again, first reduce all words to their canonical `eo` form. And once again, the problem is now much easier to think about—we just want to ensure that each word appears an even number of times in our list, since they can only be deleted in twos.

So, we count the number of times each (reduced to `eo` form) word appears using a data structure like a `dict` (in Python) or `map` (in C++). For each key-value pair in this dictionary, if the value (count) is odd, we should output another copy of the key in order to make it even.

```
from collections import defaultdict

# map i -> e and u -> o; everything else is untouched
def canon_letter(c):
    if c == 'e' or c == 'i':
        return 'e'
    elif c == 'o' or c == 'u':
        return 'o'
    else:
        return c

# apply the canon_letter transformation to each letter
def canon(w):
    return ''.join(canon_letter(c) for c in w)

n = int(input())
# defaultdict --- if a key is not found, make a new entry for it whose value is 0
freq = defaultdict(int)
for _ in range(n):
    freq[canon(input())] += 1

to_add = []
for key, value in freq.items():
    if value % 2 == 1:
        to_add.append(key)

print(len(to_add))
for word in to_add:
    print(word)

```