A

You can just an if statement with eight equality checks. We can even reduce this to four equality checks if we force c to be uppercase before performing our check.

```python
c = input().upper()
if c == 'A' or c == 'B' or c == 'K' or c == 'D':
    print('Abakoda')
else:
    print('Boring')

```

There's a nicer solution which copy pastes less, and generalizes better. Make any container with the list of Abakoda letters, and then check if c is a member of this container.

How you'd do this depends on the container used, and the syntax of your programming language. For example, here's one way to do it in Python.

```python
c = input()
if c.upper() in 'ABKD':
    print('Abakoda')
else:
    print('Boring')

```

B

We can check if a single letter is a *hiram na titik* by recycling code from Problem A.

To check if any letter of a word contains a hiram na titik, you can create a boolean flag which is “triggered” when a hiram na titik is encountered. Optionally, if one is found, you can even `break` early, since the search is already done.

```python
word = input()

has_hiram = False
for c in word:
    if c.upper() in 'CFJQVXZ':
        has_hiram = True

print('FOREIGN' if has_hiram else 'PINOY')
```

Another common approach is to wrap your logic around a *function call*, since the `return` command from always causes the rest of the function to just stop and exit immediately. This is also nice because using functions is nice, since we communicate our intent more clearly, making the code more readable.

```python
word = input()

def has_hiram(word):
    for c in word:
        if c.upper() in 'CFJQVXZ':
            return True
    return False

print('FOREIGN' if has_hiram(word) else 'PINOY')
```

In fact, this *pattern* is so common that some languages like Python and JavaScript have builtin idiomatic ways of expressing it. In Python, you could use `any`

```python
word = input()

print(
    'FOREIGN'
    if any(
        c.upper() in 'CFJQVXZ'
        for c in word
    )
    else 'PINOY'
)
```

The above code could be condensed into a one-liner, if you want. But the tutorial-writer likes liberal use of whitespace to make code more readable.

K

This problem may seem painful to implement, but with liberal use of functions, we will not just have an easy time—our code will be *readable*.

Let's break the problem down into two parts:

- Identify all the words in the sentence
- Determine which should be censored, and censor them

The second bullet is the easy part—let's throw our code for Problem B into a function, for our convenience.

```python
def has_forbidden(word, forbidden):
    return any(c.upper() in forbidden for c in word)

```

For the first bullet, let's return to the given definition: "a contiguous sequence of upper and/or lowercase English letters". In other words, punctuation chops up our sentence into chunks, and each *chunk* is a word (as indicated by the Sample I/O, this is a very *bad* definition of a word, but let's work with it).

Let's scan each character in our line, one by one. Maintain a running variable—mine is called `curr_word`—which keeps track of the consecutive letters that we've found so far. If we encounter a letter, append it to `curr_word`. If we encounter a non-letter character, then that means the group of letters in `curr_word` is a single word which had just ended.

At this point, we can store our word in some list of words—or, for this problem, we can just output it directly (after censoring the word, if necessary). In either case, don't forget to either store or output the punctuation character. (Why does the logic not break if we encounter two punctuation in a row? What happens then?)

Also, don't forget that the final word in the line might not end in punctuation—the end of that word might be signaled by the end of the entire line.

```python
bawal = input()
line = input()

curr_word = []
def flush_curr_word():
    if has_forbidden(curr_word, bawal):
        print('*' * len(curr_word), end='')
    else:
        print(''.join(curr_word), end='')
    curr_word.clear()

for c in line:
    if is_letter(c):
        curr_word.append(c)
    else:
        flush_curr_word()
        print(c, end='')

if len(curr_word) > 0:  # the final word may not have punctuation after it!
    flush_curr_word()
print()  # print the end of line character
```

Note that in this style of coding, we type out the function `is_letter(c)` *first*, believing that we can come back to it and fill in the details later. This is called *top-down design*. It's an intuitive way for humans to understand complex procedures at a higher level—focus on the main idea first, then do another pass to fill in the details.

In any case, here's how we might go back and implement that function.

```python
uppercase = 'QWERTYUIOPASDFGHJKLZXCVBNM' # or, from string import uppercase
lowercase = 'qwertyuiopasdfghjklzxcvbnm' # or, from string import lowercase
def is_letter(c):
    return c in uppercase or c in lowercase
```

D

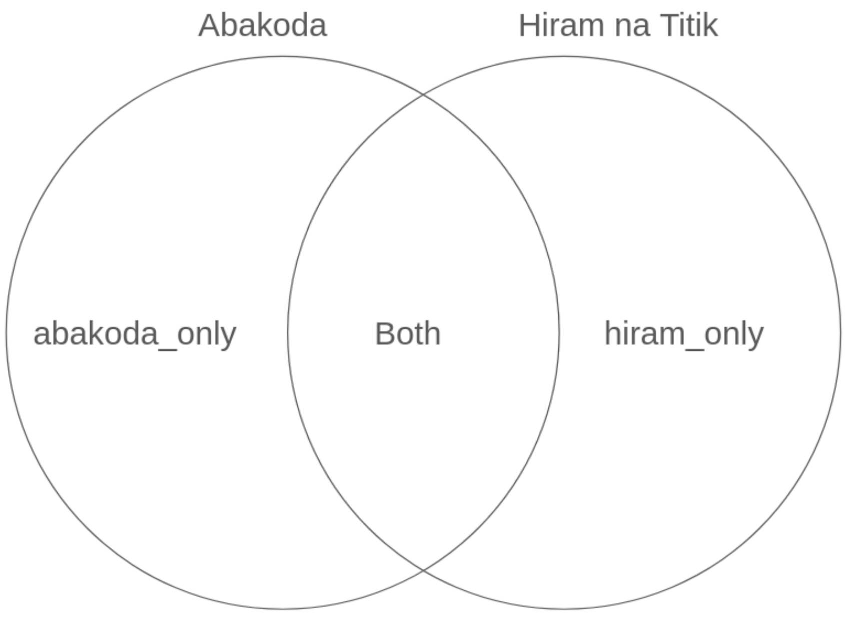
Remember that a computer program can only do $\approx 10^7$ to 10^8 operations per second—or at least, somewhere around that general order of magnitude. There are $\mathcal{O}(n^3)$ triples, and with $n = 10^5$, we definitely don't have time to go over each of them. We need to count more smartly.

Now, **if** the choices for the first, second, and third hashtag were independent of each other, then this would be straightforward. The Rule of Product from basic combinatorics gives us our answer:

$$\begin{aligned} &\#\text{ of choices for 1st hashtag} \times \#\text{ of choices for 2nd hashtag} \times \#\text{ of choices for 3rd hashtag} \\ &= |\text{Abakoda words}| \times |\text{Borrowed words}| \times |\text{Words with both}|. \\ &\text{This is incorrect} \end{aligned}$$

This is incorrect because the choices *aren't* independent of each other. We may not repeat words, therefore if we choose a word with an Abakoda letter *and* a hiram na titik for the first hashtag, it actually affects the number of options for the second and third hashtags. The problem is that some words belong to multiple sets, screwing with the independence.

So... let's make them disjoint! Let's consider a different set of sets. Consider the following Venn diagram.



Let's operate on those regions that *exclude* the intersection, which we've labeled `abakoda_only` and `hiram_only`; for convenience, let the sizes of these sets be a and h, and let the size of the intersection region be b.

Let `a_only` denote a word with an Abakoda letter *and no hiram na titik*, let `h_only` denote a word with a hiram na titik *and no Abakoda letter*, and let `both` denote a word with both.

Here are all the patterns that match with the original problem's requirements, as well as how much each case contributes to the total count.

- `#a_only #h_only #both`
  - $a \times h \times b$
- `#a_only #both #both`
  - $a \times b \times (b-1)$
- `#both #h_only #both`
  - $b \times h \times (b-1)$
- `#both #both #both`
  - $b \times (b-1) \times (b-2)$

This time, the solution is valid because a and b and h count disjoint sets, so the combi behaves in a more predictable (and "standard") way. Also, you can check that the four cases are disjoint (so no case is double-counted) and that they cover all possible valid scenarios.

```python
def has_a_letter(word, letters):
    return any(c.upper() in letters for c in word)

abakoda = 'ABKD'
hiram = 'CFJQVXZ'

n = int(input())
a, h, b = 0, 0, 0

for _ in range(n):
    word = input()
    has_abakoda = has_a_letter(word, abakoda)
    has_hiram = has_a_letter(word, hiram)
    if has_abakoda and not has_hiram:
        a += 1
    elif not has_abakoda and has_hiram:
        h += 1
    elif has_abakoda and has_hiram:
        b += 1

total = 0

total += a * h * b
total += a * b * (b-1)
total += b * h * (b-1)
total += b * (b-1) * (b-2)

print(total)

```