

## Alien Rhyme (10pts, 27pts)

### Problem

During some extraterrestrial exploration, you found evidence of alien poetry! Your team of linguists has determined that each word in the alien language has an accent on exactly one position (letter) in the word; the part of the word starting from the accented letter is called the accent-suffix. Two words are said to rhyme if both of their accent-suffixes are equal. For example, the words `PROL` and `TARPOL` rhyme if the accented letter in both is the `O` or the `L`, but they do not rhyme if the accented letters are the `R`s, or the `R` in `PROL` and the `P` in `TARPOL`, or the `O` in `PROL` and the `L` in `TARPOL`.

You have recovered a list of  $N$  words that may be part of an alien poem. Unfortunately, you do not know which is the accented letter for each word. You believe that you can discard zero or more of these words, assign accented letters to the remaining words, and then arrange those words into pairs such that each word rhymes only with the other word in its pair, and with none of the words in other pairs.

You want to know the largest number of words that can be arranged into pairs in this way.

### Input

The first line of the input gives the number of test cases,  $T$ .  $T$  test cases follow. Each test case starts with a line with a single integer  $N$ . Then,  $N$  lines follow, each of which contains a string  $W_i$  of uppercase English letters, representing a distinct word. Notice that the same word can have different accentuations in different test cases.

### Output

For each test case, output one line containing `Case #x: y`, where  $x$  is the test case number (starting from 1) and  $y$  is the size of the largest subset of words meeting the criteria described above.

### Limits

$1 \leq T \leq 100$ .

Time limit: 20 seconds per test set.

Memory limit: 1GB.

$1 \leq \text{length of } W_i \leq 50$ , for all  $i$ .

$W_i$  consists of uppercase English letters, for all  $i$ .

$W_i \neq W_j$ , for all  $i \neq j$ . (Words are not repeated within a test case.)

Test set 1 (Visible)

$2 \leq N \leq 6$ .

Test set 2 (Hidden)

$2 \leq N \leq 1000$ .

### Sample

Input	Output
4	
2	
TARPOL	
PROL	
3	
TARPOR	
PROL	
TARPRO	
6	Case #1: 2
CODEJAM	Case #2: 0
JAM	Case #3: 6
HAM	Case #4: 2
NALAM	
HUM	
NOLOM	
4	
PI	
HI	
WI	
FI	

In Sample Case #1, the two words can rhyme with an appropriate accent assignment, as described above, so the largest subset is the entire input.

In Sample Case #2, no two words can rhyme regardless of how we assign accents, because any two suffixes will differ at least in the last letter. Therefore, the largest subset is the empty one, of size 0.

In Sample Case #3, we can use the entire set of words if we accentuate `CODEJAM` and `JAM` at the `JS`, `HAM` and `NALAM` at their last `AS` and `HUM` and `NOLOM` at the `MS`.

In Sample Case #4, any two words can be made to rhyme, but always by making the accented letter the `I`. Therefore, if we add two pairs to the subset, words from different pairs will rhyme. We can, thus, only form a subset of size 2, by choosing any 2 of the input words.

## Analysis

### Test set 1

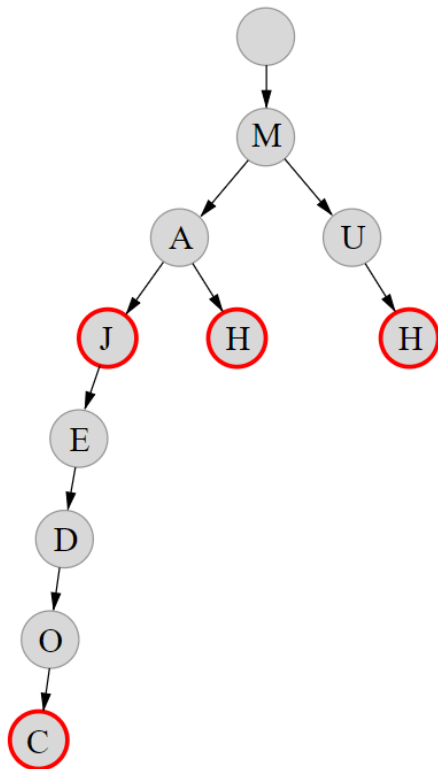
In the first test set there are only up to  $N = 6$  words with up to 50 characters in each of them. We can simply use brute force to try all ways of grouping words into pairs (allowing some words not to be in any pair – we will effectively discard these words), try all choices of an accent-suffix for every pair, and check that none of the pairs have the same accent-suffix. Finally, we choose the maximum size across all valid groupings.

### Test set 2

Let's notice how the size of an accent-suffix affects the chance of multiple words sharing it. In case we have words `CODEJAM`, `JAM`, `HUM` and `HAM`, accent-suffix `JAM` can be part of only two words, whereas a shorter accent-suffix `M` fits all four words. This leads to the following observation: for

two words that we want to pair, it is never suboptimal to choose their longest available common suffix as the accent-suffix – this way we are still making sure that they rhyme, and we are allowing shorter accent-suffixes to be used by other pairs. Notice that any other pair that could use the longer suffix can also use any shorter suffix. For example, if we want words `CODEJAM` and `JAM` rhyme, we should choose `JAM` as their accent-suffix, and allow suffix `M` to be potentially used by `HUM` and `HAM`.

In this problem it is all about common suffixes of the words. In order to better operate with word suffixes, let's actually reverse the words first (so now original word suffixes are prefixes of reversed words), and build a [trie](#) (also often called prefix tree) on the reversed words. This is how a trie containing the words `CODEJAM`, `JAM`, `HUM` and `HAM` looks:



Let's also mark the trie nodes where some of the input words end. Since we are guaranteed that all the words are unique, we can use a simple boolean flag. In the picture above, trie nodes where a word ends are marked in red.

Now we can solve the problem as follows: for a trie node  $v$ , let  $f(v)$  be the minimum possible number of unpaired words that use accent-suffixes whose reverses end in the node  $v$  or the subtree under it. The answer to the problem is then  $N - f(\text{root})$ , since  $f(\text{root})$  represents all usable accent-suffixes.

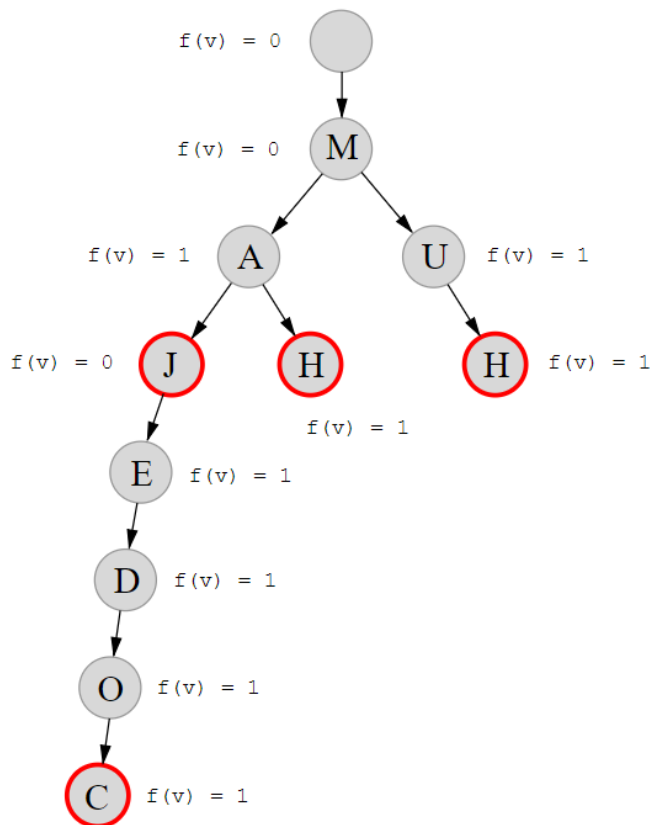
How do we calculate the values of  $f(v)$ ? If  $v$  does not have any children nodes, we set  $f(v)$  to be 1, since we know that in our trie all leaf nodes are the end of a word. If node  $v$  has children, we can calculate  $f(v)$  with the following algorithm assigning the result to  $r$ :

```

r = sum(f(c) for all c where c is a child node of v)
if node v is marked (there is a word that ends at v): r = r + 1
if v is not the root and f(v) ≥ 2: r = r - 2
  
```

First, we simply count the number of unpaired words recursively. Finally, we let two of those words be paired using the prefix that ends in the node  $v$ , which represents suffixes of original words, as the accent-suffix.

In our example trie we would get the following values of  $f(v)$ :



Proving the algorithm above correctly calculates  $f(v)$  is straightforward. First notice that only words that are represented in the trie at or below  $v$  are pairable with the set of accent-suffixes represented by  $v$  or its subtree. Then we can proceed by induction:  $f$  is pretty clearly correct for a single node tree, as there is no pairing possible. Assume by the inductive hypotheses  $f$  works correctly on all proper subtrees under  $v$ . The pairing implied by the construction of  $f(v)$  — adding any remaining pair of words to the recursive result — is valid: we are only pairing two words with the accent-suffix represented by  $v$ , and the rest is valid by the inductive hypotheses. To show that the pairing it is also of maximum size, notice that, by inductive hypothesis, there is no way to pair more than  $\sum(f(c))$  for all  $c$  where  $c$  is a child node of  $v$  words with accent-suffixes that are represented by the subtree but not by  $v$  directly. This is because words in different subtrees of  $v$  cannot be matched with a longer accent-suffix than the one represented by  $v$ , and the accent-suffix represented by  $v$  can add at most a single pair to the total.

Note how we calculate the values of  $f(v)$  in a recursive manner, and  $f(v)$  is calculated exactly once for each possible  $v$ . Since the algorithm itself takes constant time in addition to the time of the recursion, we can calculate all  $f(v)$  values in  $O(T)$  time, where  $T$  is a total number of nodes in the trie. We can bound  $T$  by the total length of all words, or by  $N \times m$  where  $m$  is the maximum word length.

Finally, there are less efficient but simpler implementations that also work. For example, sort the reversed words alphabetically and take any two adjacent words with a longest common prefix, pair them, remove them from the list, and repeat. This simple-to-implement algorithm basically constructs the same pairing our recursive formulation does. This shifts some implementation complexity onto the correctness proof. If you are faster with proofs than with code, it might be an overall gain in solving speed.