**Event:** Google Code Jam to I/O for Women 2020

**Programmer:** Destiny Stevens

**Problem:** Interleaved Output: Part 1

**PROBLEM:**

*You do not need to read the* ***Interleaved Output: Part 2*** *problem to be able to solve this problem. Both Part 1 and Part 2 have the same first two paragraphs (not including this informational text). We have underlined the critical difference between the two parts.*

On a distant moon of Jupiter, some developer conference events are about to happen! They are called IO (uppercase I, uppercase O), Io (uppercase I, lowercase o), iO (lowercase I, uppercase O), and io (lowercase I, lowercase O).

The best way to advertise an event is by using special computers that print the event's name one character at a time, with the output appearing on a digital display. Each such computer only knows the name of one event, and is programmed to print its event's name zero or more times. For example, a computer programmed to print IO twice prints an I, followed by an O, followed by an I, followed by an O, for a final string of IOIO.

You know that the conference organizers are using these computers, but you do not know how many computers are advertising each event. For each event, there may be any number of computers (including zero) programmed to print the event's name. Moreover, the computers are not necessarily all programmed to print the same number of times. For example, it is possible that there are three computers programmed to print Io once each, and one computer programmed to print Io twice.

The computers have all finished printing, but unfortunately, they all printed to the same display! Because the computers printed concurrently, event names in the final output string may be interleaved. You are considering the possible ways in which that string could have been produced.

For example, the string IiOioIoO could have been produced by two computers, as follows:

* A: programmed to print Io twice
* B: programmed to print iO twice

index: 1 2 3 4 5 6 7 8

A: I . . . o I o .

B: . i O i . . . O

string: I i O i o I o O

In this interpretation, the Io event was advertised twice, the iO event was advertised twice, and the other two events were not advertised at all.

But the string could have also been produced by three computers:

* A: programmed to print IO twice
* B: programmed to print io once
* C: programmed to print io once

index: 1 2 3 4 5 6 7 8

A: I . O . . I . O

B: . i . . o . . .

C: . . . i . . o .

string: I i O i o I o O

In this interpretation, the IO event was advertised twice, the io event was advertised twice, and the other two events were not advertised at all. Notice that this interpretation required two computers printing io; there could not have been just one computer printing io twice, because it would have had to print i twice in a row, which is not allowed.

Given a final output string, what is the maximum possible number of times that the event IO could have been advertised?

It is guaranteed that the string has at least one valid interpretation. For example, oI and IOI are not valid inputs.

**Input**

The first line of the input gives the number of test cases, **T**. **T** lines follow; each represents a single test case. Each case consists of a string **S** containing only the characters from the set I, O, i, and o.

**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 maximum number of times IO could have been advertised, as described above.

**Limits**

Time limit: 20 seconds per test set.  
Memory limit: 1GB.  
1 ≤ **T** ≤ 100.  
The length of **S** is even.  
For each prefix S' of **S**, the number of i characters plus the number of I characters in S' is not less than the number of o characters plus the number of O characters in S'.  
The number of is plus the number of Is in **S** is equal to the number of os plus the number of Os in **S**.  
(Notice that the above three conditions guarantee that there is at least one interpretation of **S** that is consistent with the rules in the problem statement.)

**Test set 1 (Visible Verdict)**

2 ≤ the length of **S** ≤ 8.

**Test set 2 (Hidden Verdict)**

2 ≤ the length of **S** ≤ 100.

**Sample**

|  |  |
| --- | --- |
| Input | Output |
| 5  IiOioIoO  IiOOIo  IoiOiO  io  IIIIOOOO | Case #1: 2  Case #2: 1  Case #3: 0  Case #4: 0  Case #5: 4 |

Sample Case #1 is the one described in the problem statement. We saw that there is an interpretation in which IO was advertised twice. There are only two Is and two Os in the string, so the answer cannot be larger than this.

In Sample Case #2, notice that it is not possible that IO was advertised twice. The only possible interpretations are as follows:

* A: programmed to print IO once
* B: programmed to print iO once
* C: programmed to print Io once

index: 1 2 3 4 5 6

A: I . . O . .

B: . i O . . .

C: . . . . I o

string: I i O O I o

or the same but with

index: 1 2 3 4 5 6

A: I . O . . .

B: . i . O . .

C: . . . . I o

string: I i O O I o

In either of these interpretations, IO was advertised only once.

In Sample Case #3, there is no possible interpretation in which IO was advertised. There must have been one computer programmed to print Io once, and either one computer printing iO twice, or two computers printing iO once each.

In Sample Case #4, notice that it is possible that I and/or O might not even show up in the string.

In Sample Case #5, an interpretation in which IO was advertised four times requires four computers, each of which was programmed to print IO once.

**ANALYSIS:**

**Test Set 1**

One way of approaching this test set is to try all ways of assigning pairs of characters in the string (an uppercase I or lowercase i followed by an uppercase O or lowercase o) to particular computers, and in each case, recursively check whether the remaining string could have been produced. If, at any point during our check, our string starts with an uppercase O or lowercase o, then the string is invalid and we can abandon that branch of the search.

For example, if we have IoiO, we can try assigning the uppercase I and the uppercase O to an IO computer, then recursively check the remaining oi and find that it's invalid, then try instead assigning the uppercase I and lowercase o to an Io computer, and so on. Finally, find the largest possible number of advertisements of IO, across all valid assignments.

**Test Set 2**

The above strategy is too slow for Test Set 2 because there are too many possible pairs to check. Let's look for a simpler strategy.

As we scan through a string, whenever we encounter an uppercase or lowercase O, we must find a previous unused uppercase or lowercase I to pair it with (implicitly claiming that a particular computer printed both of those characters). Intuitively, to find an interpretation that maximizes the number of times IO is advertised, we would like to do the following as much as possible:

* Preferentially pair an uppercase O with an uppercase I, rather than with a lowercase i
* Preferentially pair a lowercase o with a lowercase i, rather than with an uppercase I

However, we may not always be able to get what we want! For example, suppose we are scanning through the input IoiO. When we reach the lowercase o, we have no choice but to pair it with the preceding I.

We can prove, though, that if we adhere to the above preferences whenever we have a choice, we will find the correct answer. Suppose we are carrying out this method, and we reach an uppercase O — call it O1 — that we can match to either some previous unmatched uppercase I — call it Iprev — or some previous unmatched lowercase i — call it iprev. Suppose we choose to match O1 with iprev. Then we must eventually match Iprev with some later uppercase or lowercase O. (It is guaranteed that at least one of these exists because the input satisfies a balanced parentheses constraint, as outlined in the Limits section.) Call that later O O2. We will obtain 1 "point" (i.e. we will be able to claim that an IO computer advertised its event once) if O2 is uppercase, and 0 points if O2 is lowercase.

But then observe that we could have instead matched the uppercase O1 with Iprev (scoring 1 point *for sure*), and matched the lowercase iprev with the same O2. This argument holds true no matter which O2 we picked. So preferentially matching with the Iprev is no worse, and may be better.

A similar argument holds for the situation in which we reach a lowercase o that we can match to either a previous uppercase I or a previous lowercase i, and it tells us to preferentially match with the iprev.

We have covered the only two cases in which we can make a decision. Since no other strategy is better in either case, our strategy is an optimal one. It is possible that we may have a choice of e.g. two previous uppercase Is and one previous lowercase i, but then it does not matter which of the uppercase Is we pick, since whichever one we don't use will still be "previous" for the purposes of later decisions.

A simpler way to frame this strategy is as follows: scan through the string from left to right, keeping two counts: CI, the number of unmatched uppercase Is seen so far, and Ci, the number of unmatched lowercase is seen so far. When we encounter an uppercase O, score one point and decrement CI if CI is positive, or otherwise decrement Ci. When we encounter a lowercase o, decrement Ci if Ci is positive, or otherwise decrement CI. Notice that the rules of the problem guarantee that these counts will never simultaneously become zero at the time we encounter an uppercase O or lowercase o.