Code Jam 2013 - Round 1C

Analysis: Consonants

Solving the small

It cannot be simpler than trying each possible substring given the name. For a given substring, we just check if there exists **n** consecutive consonants. If it is true, we count this substring into part of the **n**-value. There are O($\mathbf{L}^2$) substrings, and it takes O(**L**) time to check for at least **n** consecutive consonants. In total each case takes O($\mathbf{L}^3$) time to solve, which is acceptable to solve the small input. This approach is, of course, not fast enough to solve the large input.

Improving the naive algorithm

In fact we can skip the linear time checking for all possible substrings. Here we assume the index is zero based. Suppose we start from the **i**-th character. We also have **c** that starts as zero. When we iterate up to the **j**-th character, if it is a consonant, we increase **c** by 1, otherwise reset it to zero. Actually **c** is the number of consecutive consonants that starts after the **i**-th character and ends at the **j**-th character. If we meet the first instance such that **c** ≥ **n**, we can conclude that every substring which starts at the **i**-th character and ends at the **k**-th character, where **k** ≥ **j**, is the desired substring. Then we know that we can add **L** - **j** to the answer, and proceed to the next starting character. This algorithm runs in O($\mathbf{L}^2$) time, which is still not sufficient in solving the large input. But the concept of computing **c** is the key to solve the problem completely.

Further improving

Let us extend the definition of **c** to every character, call it $\mathbf{c_i}$: the number of consecutive consonants that ends at the **i**-th character. For example, suppose the string is **quartz**, then $\mathbf{c_0}$ = 1, $\mathbf{c_2}$ = 0, and $\mathbf{c_5}$ = 3. We can use similar approach mentioned in the last section to compute every $\mathbf{c_i}$ in O(**L**) time. Also define a pair (**x**, **y**) to be the substring that starts at the **x**-th character and ends at the **y**-th character.

Knowing from the previous section, if we know that $\mathbf{c_i}$ ≥ **n**, then we know that substrings (**i** - $\mathbf{c_i}$ + **p**, **i** + **q**), where 1 ≤ **p** ≤ $\mathbf{c_i}$ - **n** + 1 and 0 ≤ **q** ≤ **L** - **i** - 1, are the desired substrings. It implies that there are ($\mathbf{c_i}$ - **n** + 1) × (**L** - **i**) substrings. If you proceed like this, you missed some substrings. Consider the string **axb** with **n** = 1. We see that $\mathbf{c_1}$ = 1 but we only count 2 substrings, namely **x** and **xb**. We miss the **prefix** options, namely **ax** and **axb**. It looks like we can consider the substrings (**p**, **i** + **q**), where 0 ≤ **p** ≤ **i** - **n** + 1 and 0 ≤ **q** ≤ **L** - **i** - 1. Unfortunately, in this case we may count certain substrings multiple times. Consider the string **xaxb** with **n** = 1, where we count **xax** and **xaxb** twice since $\mathbf{c_0}$ = $\mathbf{c_2}$ = 1.

To correctly count the substrings, we need to choose the appropriate range of **p**. In fact, we just need one more value: the last **j** < **i** such that $\mathbf{c_j}$ ≥ **n**. Let **r** = **j** - **n** + 2 if there is such **j**, or **r** = 0 otherwise. Then we have the right set of substrings (**p**, **i** + **q**), where **r** ≤ **p** ≤ **i** - **n** + 1 and 0 ≤ **q** ≤ **L** - **i** - 1. In fact, **r** means the longest possible prefix so that (**r**, **i** - **n**) contains at most **n** - 1 consecutive consonants and therefore we avoid repeated counting. Hence for each $\mathbf{c_i}$ ≥ **n** we count (**i** - **n** - **r** + 2) × (**L** - **i**). Summing up we have the answer. **r** is updated whenever we see that $\mathbf{c_i}$ ≥ **n** before iterating the next position. Therefore it takes constant time to update the value. Overall the running time is O(**L**), which is enough to solve the large input.

Despite the complications, the algorithm is extremely simple. The following is a sample solution:

```
def Solve(s, n):
  L = len(s)
  cnt, r, c = 0, 0, 0
  for i in range(L):
    c = c + 1 if s[i] not in "aeiou" else 0
    if c >= n:
      cnt += (i - n - r + 2) * (L - i)
      r = i - n + 2
  return cnt
```