Code Jam to I/O for Women 2016 - Code Jam to I/O for Women

Analysis: Polynesiaglot

In Small dataset 1, all of the cases have only one consonant and only one vowel available. One approach is to designate characters like `c` for the consonant and `v` for the vowel, then generate all strings of length **L** consisting of only `c`s and/or `v`s. Then, just find the subset of those strings that follow the "Any consonant in a word must be immediately followed by a vowel" rule, and find the size of that subset.

That subset can be extended to solve Small dataset 2, in which we can have multiple consonants and/or vowels. This time, think of `c` as meaning "any one of the available consonants" and `v` as meaning "any one of the available vowels". Then every valid word in the language matches exactly one of these templates, and we just have to figure out how many words match each template. For example, let's consider **C** = 2, **V** = 3, **L** = 2. Then the possible templates are `cv` and `vv`. For `cv`, there are 2 choices for the consonant and 3 choices for the vowel, so there are 2 * 3 = 6 different valid words match that template. For `vv`, there are 3 choices for the first vowel and 3 choices for the second vowel, so there are 3 * 3 = 9 words the match the second template. So the total number of valid words is 6 + 9 = 15.

What about the Large dataset? For large values of **L**, there are far too many different templates to generate using the above method.

If you looked carefully at the results for Small dataset 1, you might have noticed a pattern: for **L** = 1, 2, 3, 4, 5, 6, 7..., the results are 1, 2, 3, 5, 8, 13, 21... That's exactly the Fibonacci sequence, in which each term in the sequence equals the sum of the two previous terms. Can we frame this problem in a similarly recursive way?

We will lay out one thorough approach to reaching a recursive formula. Consider any valid word of length X in the language. Imagine removing the first letter. Then the remainder must also be a valid word. This is because removing the first letter can't have created a situation in which a consonant isn't followed by a vowel. (If we'd removed the last letter instead, the leftovers might have ended in a consonant!) So any valid word in the language is either a vowel or a consonant, followed by either some other valid word in the language, or the empty word. (That last part is needed to account for valid words of length 1.)

Let's define $W_C(X)$ as the number of valid words of length X starting with a consonant, $W_V(X)$ as the number of valid words of length X starting with a vowel, and $W(X)$ as the total number of valid words of length X. Then we can write:

$W_C(X) = \mathbf{C} * W_V(X\text{-}1)$
(For any valid word of length X starting with a consonant, the rest of the word can be any valid word of length X-1 starting with a vowel. There are **C** different consonants that we can choose to put in front of any of those words.)

$W_V(X) = \mathbf{V} * W(X\text{-}1)$
(For any valid word of length X starting with a vowel, the rest of the word can be any valid word of length X-1. There are **V** different vowels that we can choose to put in front of any of those words.)

Substitute the second equation into the first to get:

$W_C(X) = \mathbf{C} * \mathbf{V} * W(X\text{-}2)$

Finally, $W(X)$ is just $W_V(X)$ plus $W_C(X)$:

$W(X) = \mathbf{V} * W(X\text{-}1) + \mathbf{C} * \mathbf{V} * W(X\text{-}2)$

Since this is a recursive definition, we need some base cases:

$W(0) = 1$
(We must treat the empty word as a valid basis for building additional words.)

$W(1) = \mathbf{V}$
(There are **V** valid words of length 1, since the word can only be one of the **V** different vowels, and not one of the consonants.)

You can turn that last equation and the two base cases directly into code. The biggest potential problem is repeating the same sub-calculations over and over again. One way to avoid that is via memoization, a form of dynamic programming: create a structure to hold the results of individual calculations like $W(3)$. Then, before performing any calculation, check the array to see if you already have that result. If you do, just use that; if you don't, then compute the result and store it in the array.

Alternatively, you can avoid storing so many values by starting from $X = 0$ and working up to $X = \mathbf{L}$, instead of working backwards from $X = \mathbf{L}$. Since $W(X)$ only depends on $W(X\text{-}1)$ and $W(X\text{-}2)$, you can compute $W(2)$ using $W(1)$ and $W(0)$, then compute $W(3)$ using $W(2)$ and $W(1)$ (note that we no longer care about $W(0)$ at this point), and so on. This approach only keeps track of three values at a time, whereas the memoization approach requires O(**L**) memory. (You can use a similar approach to calculate any Fibonacci number using only two variables.)

What about the requirement to provide the remainder modulo $10^9 + 7$? This is a common situation in programming contests. In languages in which you must work with numbers of fixed sizes, you can take the result of every intermediate operation (addition or multiplication) modulo $10^9 + 7$. You may be tempted to use a language like Python that will handle very large numbers, and perform the modulo operation once at the end, but this is not a good idea in general — large numbers in intermediate calculations can slow algorithms down considerably. In this case, we'd go from O(**L**) to O($\mathbf{L}^2$).

Note that if we have only one vowel and one consonant, as in Small dataset 1, then the last equation reduces to $W(X) = W(X\text{-}1) + W(X\text{-}2)$. This is exactly the recurrence for the Fibonacci sequence, which explains the results from Small dataset 1.

The solution we have presented requires O(**L**) operations (i.e., it takes time proportional to **L**). A better solution can take advantage of the fact that the **L**-th term of a sequence defined by a recurrence relation like the the definition we outlined above can be calculated by using matrix exponentiation with only O(log **L**) matrix multiplications. Since in this case the total size of the matrix is bounded, that yields an O(log **L**) algorithm that solves the problem. We leave the details as an exercise for the reader.