Mod Powers

Subtask 1

Alice's password must look like some alternating series of consonants and vowels. Recall that there are 21 consonants and 5 vowels.

There are two cases to consider. In everything that follows, let `a` represent "any vowel" and let `b` represent "any consonant".

If n is even, then there are two possible "shapes":

```
ababab
bababa
```

which we see depends on if our first letter is a consonant or vowel. In either case, the solution is the same: we need to independently choose values for each of the $n/2$ consonants and $n/2$ vowels. Thus, the formula for even n is:

$$5^{n/2} \times 21^{n/2} + 21^{n/2} \times 5^{n/2},$$

or

$$2 \times (21 \times 5)^{n/2}$$

If n is odd, then there are still two possible "shapes":

```
abababa
bababab
```

which again depends on if our first letter is a consonant or vowel.

- If the first letter is a consonant, then choose which of the 21 consonants it should be.
- If the first letter is a vowel, then choose which of the 5 vowels it should be.

Now we just need to decide the remaining $n - 1$ letters. In either case, we end up with $(n - 1)/2$ consonants and $(n - 1)/2$ vowels whose values we must independently decide. Thus, the formula for odd n is:

$$21 \times 5^{(n-1)/2} \times 21^{(n-1)/2} + 5 \times 21^{(n-1)/2} \times 5^{(n-1)/2},$$

or

$$(21 + 5) \times (21 \times 5)^{(n-1)/2}.$$

It may be a bit tedious, but you can still manually compute the value by hand when $n = 10$, using this formula. Just use a calculator to speed things up.

Subtask 2

Directly implement this formula in code, where we implement exponentiation as repeated multiplication—compute a^b by multiplying a to itself b times. Note that because of combinatorial explosion, the raw answer is going to be huge—so you must take modulos at each intermediary step.

```
# pseudocode

n = 13**7
MOD = 10**9 + 7

# n is odd here
ans = 21+5
for (n-1)/2 times:
    ans *= 21*5
    ans %= MOD
print(ans)
```

Even a slow language like Python can do on the order of $\approx 10^7$ operations per second, so this should terminate within a few seconds for the n in subtask 2.

Subtask 3

If n is very very large, then "literally do something $n/2$ times" is way too slow. We need a faster exponentiation algorithm so that we can use our magic formula.

If you Google "fast exponentiation algorithm", you'll find many results for algorithms that achieve the result in only $\sim \log_2 n$ multiplications. Such algorithms usually (explicitly or implicitly) leverage the binary representation of the exponent. The Wikipedia article calls the technique, "exponentiation by squaring".

The author would also like to suggest his writeup here which gives a recursive formulation of the algorithm.

Pick your favorite algorithm and implement it, and you will solve subtask 3 in a fraction of a second.

Subtask 4

Dealing with power towers

Let's suppose that we already have an algorithm that allows us to compute $a^b \bmod m$ in $\sim \log_2 b$ steps. Unfortunately, the "power tower" in subtask 4 is absolutely massive, and even $\log_2\left(7^{7^{7^{2023}}}\right)$ is astronomically large.

We need a way of trimming the size *of the exponent.*

One classic approach is to use **Euler's Theorem**. Let a and m be coprime integers. Then,

$$a^{\varphi(m)} \equiv 1 \pmod{m}$$

where φ is Euler's Totient function, and $\varphi(m)$ counts the number of positive integers less than or equal to m which are coprime to it. There are many proofs of Euler's Theorem online which you can refer to.

Let $b \geq \phi(m)$. Euler's Theorem tells us that:

$$a^{b+\varphi(m)} \equiv a^b \equiv a^{b-\varphi(m)} \pmod{m},$$

which inductively means that,

$$a^b \equiv a^{b+k\varphi(m)} \pmod{m},$$

for any non-negative integer k such that $b + k\phi(m) \geq 0$.

But since $b \bmod \varphi(m) = b - \lfloor b/\varphi(m) \rfloor \varphi(m)$, we conclude that:

$$a^b \equiv a^{b \bmod \varphi(m)} \pmod{m}.$$

Returning to power towers, that means, for example:

$$a^{b^{c^{d^e}}} \equiv a^{b^{c^{d^e}} \bmod \varphi(m)} \pmod{m}.$$

In other words, we know that we can compute $a^{b^{c^{d^e}}} \bmod m$ efficiently if we know how to compute $b^{c^{d^e}} \bmod \varphi(m)$ efficiently. But that's just a slightly smaller power tower problem! So, recursively, we can just use the same trick again![1] For example here, you can evaluate $b^{c^{d^e}} \bmod \varphi(m)$ by evaluating $c^{d^e} \bmod \varphi(\varphi(m))$, and so on.

Repeatedly apply this Euler's Theorem trick to shave levels off your power tower until you hit a case where the exponent is small enough that a fast exponentiation algorithm can do the trick.

[1] You have to be a bit careful when considering the cases where your base is not coprime to your modulus

Dividing by 2

Going back to our problem, recall that we want to compute (since $n = 7^{7^{7^{2023}}}$ is odd):

$$26 \times 105^{(n-1)/2} \pmod{p},$$

where $p = 10^9 + 7$. From the power-tower trick we just discussed, we know that this task can be done if we can compute,

$$(7^{7^{7^{2023}}} - 1)/2 \pmod{\varphi(p)},$$

which (as we said) is done by repeating the Euler's Theorem trick. There's one final hiccup we need to address—$\varphi(10^9 + 7)$ is **even**, meaning that 2 does not have a unique multiplicative inverse modulo $\varphi(p)$. How do we divide by 2 now?

We need one more trick. Suppose a is divisible by some d, and

$$a \equiv r \pmod{dm}$$

Then, we can show that r is also divisible by d, and that

$$\frac{a}{d} \equiv \frac{r}{d} \pmod{m}.$$

All this means for us is that we need to compute:

$$7^{7^{7^{2023}}} - 1 \pmod{2\ \varphi(p)},$$

which we do using already-established power tower techniques. *This result* can be divided by 2 to get what we wanted.

Proof. Apply the division algorithm where we divide a by dm. There is a unique pair of integers q and r such that $0 \leq r < dm$ and

$$a = (dm)q + r.$$

Note that dmq is divisible by d; so if a is divisible by d, then the sum on the RHS must also be divisible by d, and so r must be divisible by d as well.

Dividing both sides by d:

$$\frac{a}{d} = mq + \frac{r}{d}.$$

and here, $0 \leq \frac{r}{d} < m$.

Now, apply the division algorithm where we divide $\frac{a}{d}$ by m. There is a **unique** pair of integers q' and r' such that $0 \leq r' < m$ and

$$\frac{a}{d} = mq' + r'.$$

But note that $\left(q, \frac{r}{d}\right)$ satisfies these criteria for (q', r'); by the uniqueness of (q', r'), we therefore conclude that $r' = \frac{r}{d}$, which is what we wanted to show. ■

Computing totients

One standard way of computing the totient function of large integers is to use the fact that the totient function is *multiplicative.* You can do the following steps to evaluate $\varphi(m)$:

- First, prime factorize $m = p_1^{e_1} p_2^{e_2} \dots p_l^{e_k}$. There are simple factorization algorithms that run in $\sim \sqrt{m}$ steps, and that's fast enough for our purposes.
- Now, φ is multiplicative, meaning

$$\varphi\left(p_1^{e_1} p_2^{e_2} \dots p_l^{e_k}\right) = \varphi\left(p_1^{e_1}\right) \varphi\left(p_2^{e_2}\right) \dots \varphi\left(p_k^{e_k}\right)$$

- Finally, the totient function is easy to compute for prime powers. By straightforward combinatorics, you can show that $\varphi(p^e) = p^e - p^{e-1}$, when p is a prime and e is an integer ≥ 1. So, use this formula for each prime power, then multiply the results.

There are many explanations online for why φ is multiplicative (or what it means for a function to be multiplicative, in general).

An alternatively *incredibly low-effort* solution is to realize that for this problem, we only need to evaluate $\varphi(p)$, $\varphi(2\varphi(p))$, and $\varphi(\varphi(2\varphi(p)))$. Since you only need to evaluate the totient function at a handful of points, you could also just ask WolframAlpha to perform these computations for you.