Editorial 5: Big Ben's Jenga Bricks
4+1=5 Points

Problem ID: `jenga`
Rank: `2+3`

Main Test Set

The main test set can be solved by naively calculating the answer using some loops and properties of modular arithmetic.

Counting

A Jenga tower of size $3 \times 3 \times h$ is composed of h different layers of size $3 \times 3 \times 1$. Every layer is independent from every other layer, as each must be made using exactly 3 horizontal bricks. This means that there are only two ways to build each floor: one along the direction of the x-axis and one perpendicular to it. To build a tower, we choose an orientation for each floor, so by the product rule of counting, there are 2^h ways to build a tower of height h.

We're interested in counting the number of ways of building towers using at most $\mathbf{N}$ bricks. Since each layer requires 3 bricks, we can only make towers of height at most $\lfloor \frac{\mathbf{N}}{3} \rfloor$ ($\frac{\mathbf{N}}{3}$ rounded down). To get our final answer, we can simply sum up 2^h for every value of h between 1 and $\lfloor \frac{\mathbf{N}}{3} \rfloor$.

However, we're not asked to find the number of towers. We're asked to find it modulo 3359232.

Modular Arithmetic

While it may be tempting to perform all the multiplications and additions upfront and then perform a single modulus at the end, you'll run into integer overflow if you're using Java or C++, giving wrong answers. Python doesn't have this issue due to having arbitrary precision integers (but will run into time limit issues in the bonus instead). We need a way to find an answer using small, fixed-size integers. Thankfully, there are a few rules of modular arithmetic that can help us out:

$$(a + b) \mod m = (a \bmod m + b \bmod m) \mod m$$
$$(a \cdot b) \mod m = (a \bmod m \cdot b \bmod m) \mod m$$

In other words, instead of doing all the arithmetic first and modulus at the end, we can perform a modulus after every intermediate arithmetic calculation to still get the same answer. This way, our answer after intermediate calculations will always be less than 3359232. When multiplying, our answers will always be less than 3359232^2 which is small enough if we use a 64-bit integer type (`long` in Java, `long long` in C++) so as to not have to worry about overflow!

Computing the Final Answer

To get our final answer, we loop a variable h over every possible valid tower height, from 1 to $\lfloor \frac{\mathbf{N}}{3} \rfloor$. For each of these heights, we can compute 2^h by repeated multiplication in a for loop while modding the result each time, and then adding it to another variable storing the final answer while modding the answer each time. The expression of our final answer is:

$$\sum_{h=1}^{\lfloor \frac{\mathbf{N}}{3} \rfloor} 2^h \mod 3359232$$

Multiplying, adding, and modulo on fixed-size integers takes $O(1)$ time. Computing 2^h takes h multiplications, and $h \leq \mathbf{N}$, so computing each tower takes $O(\mathbf{N})$ time. There are $\lfloor \frac{\mathbf{N}}{3} \rfloor$ floors to compute in total, so the overall runtime is $O(\mathbf{N}^2)$.

Bonus Test Set

With $1 \leq \mathbf{N} \leq 10^{18}$, the $O(\mathbf{N}^2)$ algorithm for the main test set is too slow. We'll make a mathematical optimization followed by a computational optimization to speed things up.

Geometric Series

Notice that the series we are attempting to compute is a geometric series with $a = 2$ and $r = 2$. By the geometric series formula, this yields:

$$\sum_{h=1}^{\lfloor \frac{\mathbf{N}}{3} \rfloor} 2^h = \frac{2(2^{\lfloor \frac{\mathbf{N}}{3} \rfloor + 1} - 2)}{2 - 1} = 2^{\lfloor \frac{\mathbf{N}}{3} \rfloor + 2} - 2$$

So we only really need to compute one exponent instead of a whole series. This cuts our runtime down to $O(\mathbf{N})$. Still not fast enough, but a lot better.

Exponentiation by Squaring

We can implement fast modular exponentiation by using the binary exponentiation algorithm. This cuts our runtime down to $O(\log(\mathbf{N}))$, which is fast enough to pass.

Python has the built-in function `pow(base, exp, mod)` that implements this under the hood, allowing for a very easy one-liner solution. No such function is provided by the libraries of Java and C++, however, so Java and C++ users would need to implement this themself or copy an implementation published online.

Design Notes

Conception

This problem was conceived after it's much harder variant, `benga` was fully written and implemented. We thought `benga` would be a bit lonely sitting by itself in a veil of black at the end of the contest, so we created a more accessible easier version to get contestants hyped for the big challenge! Plus, we thought `jenga` was an excellent opportunity to re-use the art assets that Kaashvi had already made for `benga`.

A Dilemma

An interesting dilemma came up shortly after this problem's idea was suggested. Python users can easily solve this using the third parameter of the builtin `pow` function, while Java and C++ programmers would struggle more. Is this acceptable? Is this fair?

I'm (alfphaderp) personally of the opinion that CALICO should be a contest that promotes programming multi-lingualism. It doesn't seem right to constrain our problem designs to only target users of a specific language. We already support multi-lingualism more than many other contests by thoroughly testing all problems rank 3 or under using all programming languages. We additionally compensate by giving Java submissions 2x the time limit and Python submission 4x the time limit. Is it so bad that some problems naturally favor some languages over others?

In the end, we discovered through testing that the final bonus test set of another problem on this contest, `subway` was quite tough to get for Python users due to lacking a certain data structure provided in the standard libraries of Java and C++. This made it feel more acceptable to keep the bonus of `jenga` as is in order to compensate.

But we're curious to hear what our contestants think! Feel free to let us know in the feedback form or message us in the CALICO community discord server!

Problem Credits

- Idea: alfphaderp, nacho
- Problem Statement: nacho
- Art: Kaashvi
- Implementation: nacho
- Testing: JJ, voidcs, Justin
- Editorial: rohit, alfphaderp