Code Jam 2020 - Round 2

Analysis: Incremental House of Pancakes

Test Set 1

Test Set 1 is small enough that we can simulate the process. Since each step removes at least one pancake from a stack, this takes at most **L** + **R** operations. One thing we can notice is that because the i-th step removes i pancakes, the first i steps together remove i × (i + 1) / 2 pancakes. This means that the number of steps is actually bounded by 2 × sqrt(**L** + **R**), which means the simulation algorithm is also O(sqrt(**L** + **R**)). It would work for limits that are much larger than the ones in this test set!

Test Set 2

Unfortunately, O(sqrt(**L** + **R**)) is still too slow for the 10^{18} limits in Test Set 2, especially with 1000 cases to go through!

We can simulate multiple steps at a time by noticing that there are two distinct phases of the process. The first phase uses a single stack: the one that starts out with more pancakes. The second phase begins when that stack has a number of pancakes remaining that is less than or equal to the number in the other stack. Notice that if the left stack is the one we serve from in phase 1, it is possible that it is also the first one to be used in phase 2. It is also possible that no customer is served in either phase.

We may serve a lot of customers in phase 1 depending on the difference in size between the stacks at the beginning. If we serve i pancakes from one stack, we remove i × (i + 1) / 2 pancakes from it, so we can efficiently calculate how many customers we can serve in phase 1 by finding the largest i_1 such that $i_1 \times (i_1 + 1) / 2$ is less than or equal to the difference in number of pancakes between the two stacks at opening time. We can calculate that either by solving a quadratic equation and rounding carefully, or by using binary search.

The second phase is where the magic happens. Let us say that when serving customer i, stack X is used and Y is not, but then we serve customer i + 1 from stack Y. Therefore, stack X lost i pancakes and stack Y lost i + 1 pancakes. Since X was used instead of Y for customer i, X must have had no fewer pancakes than Y had at that time. Since X lost fewer pancakes than Y, X must have had more pancakes than Y after we served customers i and i + 1. This means if we ever use two different stacks in the order (X, Y), we must use X next. And, using the same reasoning with the roles reversed, we have now have used (Y, X) most recently, so we will use Y next, and so on. So, once we have used both piles, we always go on alternating between them.

We can use this observation to efficiently determine what happens in phase 2. After updating the original totals by subtracting the pancakes served in phase 1, we know which stack is used first in phase 2. The first stack will be used to serve customers $i_1 + 1$, $i_1 + 3$, $i_1 + 5$, ... which means that if it is used for c_1 customers, a total of $(i_1 \times c_1) + (c_1)^2$ pancakes are served from it. At this point, we know the value of i_1, so once again, we can calculate c_1 by solving a quadratic equation or binary search. The other stack is similar, since it will be used to serve customers $i_1 + 2$, $i_1 + 4$, $i_1 + 6$, ... so if it is used for c_2 customers, a total of $((i_1 + 1) \times c_2) + (c_2)^2$ pancakes will be served from it. So the final number of customers served is $i_1 + c_1 + c_2$. The total numbers of pancakes served from each stack come from the quantities in phase 2, with the quantity from phase 1 added to whichever stack was first.

If we use binary searches, each phase requires O(log(**L** + **R**)) time. If we directly solve the quadratic equations, each phase is actually constant time. Either is fast enough for the limits of this problem.

Notice that solving quadratic equations may be harder than usual, since typically that involves computing some square roots. While most languages provide a way to do that, they do it with double precision floating point, which does not have enough precision for this problem and can lead to off-by-one errors. We should either compute square roots directly on integers (by binary searching for the answer, for example) or use the built-in function, and then check the returned value and other values in its vicinity to find the correct rounded result.