Code Jam 2010 - Qualification Round

Analysis: Theme Park

At first glance, this problem appears to be a straightforward simulation: keep adding groups until you run out space on the roller coaster (or run out of groups), then re-form the queue and start again. Repeat this **R** times and you're done.

For the Small, that was enough; and we got more than a few questions during the contest from contestants thinking that they were unable to solve the Large because their computers were so slow. The fact is that, with limits so large -- up to 10^3 groups queuing for 10^9 rides -- you need to come up with a smarter algorithm to solve the problem, since that one is O(NR).

Optimization One

When you're sending the roller coaster out for 10^9 rides, you've got to expect that a few of them are going to be the same. If you store the queue of groups as an unchanging array, with a pointer to the front, then every time you send out a ride s, you could make a note of where it ended, given where it started. Then the next time you see a ride starting with the same group in the queue, you can do a quick lookup rather than iterating through all the groups.

That speeds up the algorithm by a factor of 10^3 in the worst case, leaving us with O(R) operations. There are some other ways of speeding up the calculation for any given roller coaster run: for example, you could make an array that makes it O(1) to calculate how many people are in the range [group_a, group_b] and then binary search to figure out how many groups get to go in O(log(N)) time. That gives a total of O(R log N) operations.

Optimization Two

As we observed in Optimization One, you're going to see a lot of repetition between rides. You're also going to see a lot of repetition between groups of rides. In the example in the problem statement, the queue was made up of groups of size [1, 4, 2, 1]. 6 people get to go at once. Let's look at how the queue changes between rides:

```
1, 4, 2, 1  [5]
2, 1, 1, 4  [4]
4, 2, 1, 1  [6]
1, 1, 4, 2  [6]
2, 1, 1, 4  [4]
4, 2, 1, 1  [6]
1, 1, 4, 2  [6]
```

As you may have noticed, there's a *cycle* of length three: starting from the second run, every third queue looks the same. We make 16 Euros when that happens, which means we'll be making 16 Euros every 3 runs until the roller coaster stops rolling.

So if the roller coaster is set to go 10^9 times: the first time it makes 5 Euros; then there are 999999999 runs left; and every three of those makes 16 Euros. 3 divides 999999999 evenly -- if it didn't, we'd have to do some extra work at the end -- so we make 5 + (999999999 / 3 * 16) = 5333333333 Euros in total.

It turns out that a cycle *must* show up within the first N+1 rides, because there are only **N** different states the queue can be in (after **N**, you have to start repeating). So you only have to simulate **N** rides, each of which takes O(N) time in the worst case, before finding the cycle: that's an $O(N^2)$ solution.

Optimization Three

Either of the optimizations above should be enough. But if you're a real speed demon, you can squeeze out a little more efficiency by combining the binary search that we mentioned briefly in Optimization One with the cycle detection from Optimization Two, bringing our running time down to O(N log N). An alternate optimization can bring us down to O(N); we'll leave that as an exercise for the reader. Visit our Google Group to discuss it with the other contestants!