Code Jam 2010 - Round 3

Analysis: Hot Dog Proliferation

Background

This analysis will have nothing to do with hot dogs. Instead of a long street with billions of corners, let's think of the line of integers; instead of vendors, let's think of chips -- after all, chips are much easier to maneuver than real people with hot dog stands!

We denote the number of total chips by `n`. Also, let's call a configuration *stable* if no two chips occupy the same integer point.

If you play around with the game for a while or if you have good (and brave!) intuition, you might realize that the problem statement is a little misleading. It turns out that no matter which move you do at each step, the final configuration, as well as the total number of moves you need to perform, will always be the same.

Indeed, this is a famous theorem for "chip-firing games", and our scenario is a special kind of chip-firing game. Intuitively, the reason why your choices don't matter is that (a) if you ignore a move now, you will still have to do it later, and (b) one move will not change the effect of another move down the line. This means that while you can control the *order* of moves, you will always do the same set of moves in the end, and they will always have the same effect.

This observation is enough to solve the small input. Just keep doing moves until the configuration stabilizes, and count how long it took. For the large input though, more insight is required. A configuration might require over 10^13 moves to stabilize, so simulating them one at a time is out of the question. The obvious optimization is to do several moves at once for very large piles. Surprisingly however, this does not help very much.

There are a few different ways to proceed, and we will discuss two of them.

Preliminary Observations

One very useful way of understanding this game is in terms of *invariants*. The first of these is pretty obvious, but the other requires either some special insight or some experience to see. In each move, we take two chips at some position `x` and send them to positions `x-1` and `x+1`. Notice that:

$$(x-1) + (x+1) = x + x$$
$$(x-1)^2 + (x+1)^2 = x^2 + x^2 + 2.$$

This immediately leads to the following two observations:

Observation 1. The sum of the positions of all the chips never changes.

Observation 2. The sum of the squared positions of all the chips increases by 2 during each move.

So how do we use these observations? They aren't *necessary*, but they will have their uses as you will see. The former one will help us quickly construct a configuration with certain known properties from the initial configuration (more on this later). And with the latter observation, computing the number of steps becomes the same task as constructing the final configuration.

For example, using Observation 2, we can easily estimate that the number of steps could be on the order of n^3, thereby verifying that straightforward simulation really is hopeless.

Adding one chip

One good approach is to add chips one at a time, at each step doing enough moves to completely stabilize the configuration. The question is: how do we do this last part efficiently? So let's consider adding a chip to a stable configuration.

If the new chip arrives at a position where there was no chip before, we are done. Otherwise, it lands on a segment, and the picture looks something like this:

```
             *
?????????.***************.????????
```

The two "`.`"s represents empty positions. If you play around with a couple examples, you should be able to see that the ending result will always be two segments, one starting from the position of the left "." in the picture, and the other ending at the position of the right ".". We might also view the result as a single segment with a hole. Furthermore, you might also realize that if there were `A` points to the left of our new chip in the original configuration, and `B` points to the right, then the two new segments will have lengths `B+1` and `A+1` respectively, and the total number of moves required will be `(A+1)*(B+1)`.

We could also have computed the position of the hole using Observation 1. The sum of the positions in the initial configuration is (1+2+...+15)+4, and we know in the new configuration that the sum is (0+1+...+16)-H, where H is the position of the hole. Therefore, H must be 12. The final picture is

```
?????????************.****????????
```

We could then use Observation 2 to easily determine how many moves were required to get here.

So here is one possible solution to the problem. Add the chips one by one. At each stage, we have up to `n` disjoint segments. If the new chip lands on an unoccupied position, it forms a segment unto itself; otherwise, it transforms one segment into two as described above. In either case, the new segments might touch the ones to their left and/or right, and we merge them if that happens.

All that's left is to figure out how to store these segments in your program. If you are clever, you might realize that if we add the chips from left to right, then each new chip will always be on or next to one of the last two segments. You could then use a stack to store all the segments -- all the operations will be on the top two elements of the stack. This approach gives an `O(n)` solution. If you missed this last insight, you could also use a binary search tree (e.g. an STL set) to get an `O(n log n)` solution.

Adding one pile

In our problem, we have `C` piles of chips, and usually `C` is much smaller than `n`. We now sketch a lightning-fast solution that runs in `O(C)` time. This level of insight is not necessary to solve the problem, but it's still pretty interesting. As you will see, it is essential to understand the details of the above `O(n)` solution.

Instead of adding one chip at a time, we will try to process all the chips from a single position at the same time.

First let's resolve the case when there is only one pile of n chips at position x. By symmetry and the discussions in the previous section, it is easy to see that the stable configuration is a segment centered at x if n is odd; and a segment centered at x with a hole in the center if n is even.

Let's define an *H-segment* to be a segment with a hole. It is a tuple (x, y, z), where $x < y \leq z$, representing a segment of chips from position x to position z, inclusive, but with position y empty. Note that, when $y = z$, the hole is at the very end, and it is actually a normal segment.

Our solution adds the piles one by one. And we keep a stack of existing H-segments from the left to the right. When a new pile comes, it is transformed into a new H-segment. If the H-segment does not overlap with any existing H-segments, we are done. Otherwise, it overlaps with the topmost H-segment in the stack; that is, it creates some positions with two chips. But using the observations from the last section, we know that if we resolve the conflicts one at a time, we will always have at most one hole. That means the result will be another H-segment. If the new one overlaps with the current top H-segment in the stack, we continue with the same resolving process. We do this until the stack is empty, or the H-segment is disjoint from the top of the stack. Then we push the new one and proceed to the next pile.

It remains only to explain how to compute a new H-segment quickly. And the answer is: just use Observation 1 again! When resolving two H-segments, we know S -- the sum of the positions in them; we also know the total number of chips K, so (remember the hole), $z = x+K$. We need to decide the start position x. Depending on y, the sum S satisfies

$$K(2x + K - 1) / 2 \leq S < K(2x + K + 1) / 2$$

There is a unique x satisfying this, and it can be solved in constant time. We can then find y exactly like we did in the $O(n)$ solution.

More Information

If you liked this problem, you might also enjoy reading the following classical paper on chip-firing games:
- Anders Björner, László Lovász, and Peter Shor *Chip-firing games on graphs*. European Journal of Combinatorics, Volume 12 , Issue 4 (July 1991).