Daily Coding Problem #225

Problem

This problem was asked by Bloomberg.

There are `N` prisoners standing in a circle, waiting to be executed. The executions are carried out starting with the kth person, and removing every successive kth person going clockwise until there is no one left.

Given `N` and k, write an algorithm to determine where a prisoner should stand in order to be the last survivor.

For example, if `N = 5` and `k = 2`, the order of executions would be `[2, 4, 1, 5, 3]`, so you should return `3`.

Bonus: Find an `O(log N)` solution if `k = 2`.

Solution

The most straightforward approach is to store the prisoners as a list from `1` to `N`. We can simulate the removal of the first prisoner by popping the element from the list at the `k - 1`th place. Since we have just removed a prisoner, we only need to increment the index `k - 1` to find the next prisoner to remove. This process can be continued, wrapping around our list of remaining people if necessary, until there is only one prisoner left.

```
def last_one_standing(n, k):
    people = range(1, n + 1)
    index = k - 1
    while n > 1:
        people.pop(index)
        index = (index + k - 1) % len(people)
        n -= 1
```

```
    return people[0]
```

Since popping from a random index in a list is `O(N)`, and we are doing this `N` times, this solution is actually $O(N^2)$!

One way to improve the efficiency is by changing our data structure. By using a deque, we can rotate our list in `O(k)` time, in order to only pop elements at the end. In this way, eliminating each prisoner only takes `O(k)`, and so the whole algorithm will be `O(k * N)`.

```
from collections import deque

def last_one_standing(n, k):
    people = deque(range(1, n + 1))
    while n > 1:
        people.rotate(-k)
        people.pop()
        n -= 1
    return people[0]
```

To improve this even further, we can consider a recursive solution.

Note that executing one person reduces the number of prisoners by one, and bumps the last survivor's index by k (modulo the number of prisoners). So for example, in the five-prisoner problem given above, once we have removed 2, our problem is equivalent to one in which there are only four prisoners, and our starting index is 3. In other words, our recursive relationship is as follows:

```
last_one_standing(n, k) = (last_one_standing(n - 1, k) + k) % n
```

As a result, we can continually call our function on smaller and smaller numbers of prisoners, updating the current index each time. When we reach the point where there is only one prisoner left, we increment the index a final time and return.

```
def helper(n, k):
    if n == 0:
        return 0
    else:
        return (helper(n - 1, k) + k) % n

def last_one_standing(n, k):
    people = range(1, n + 1)
    return people[helper(n, k)]
```

This solution is `O(N)`, since we call our helper function `N` times before reaching the base case.

Finally, with some math we can solve the case where `k = 2` in only `log N` time.

Let's take a look at the example problem. In the first pass around the circle, we eliminate prisoners 2, and 4. At this point, prisoner 1 is still in position 1, prisoner 3 is in position 2, and prisoner 5 is in position 3. In other words, when there are an odd number of prisoners, after one round of executions, the prisoner at index n will correspond to the previous's round prisoner at index `2 * current_index + 1`. We can state the relationship for even and odd numbers of prisoners as follows:

- if `N` is even: `last_one_standing(2 * n) = 2 * last_one_standing(n) - 1`
- if `N` is odd: `last_one_standing(2 * n + 1) = 2 * last_one_standing(n) + 1`

Rearranging our terms, we come to the following recursive `O(log N)` solution:

```
def last_one_standing(n):
    if n == 1:
        return 1
    if n % 2 == 0:
        return 2 * last_one_standing(n / 2) - 1

    else:
        return 2 * last_one_standing(n / 2) + 1
```