Code Jam 2017 - Qualification Round

Analysis: Bathroom Stalls

Test set 1

For test set 1, the limits are small enough that you can just simulate the rules outlined in the statement. Most implementations of a simulation will run in O(**NK**) time and thus finish immediately, but even a slow O($\mathbf{N}^2\mathbf{K}$) implementation like "try every possible stall for the next person, and for each empty stall run a loop for each side to check for the closest neighbors" will most likely finish in time.

For test sets 2 and 3, however, something quadratic in the number of stalls won't cut it, so we have to do better.

Test set 2

The critical observation to jump from test set 1 to test set 2 is that only the number of consecutive runs of empty stalls matters at any given time. The next person always chooses the middle stall or the left of the two middle stalls of a longest subsequence of consecutive empty stalls. Moreover, the output format already hints at this: even if you were to choose the rightmost of a set of two middle stalls, or a longest run of stalls other than the leftmost one, the answer would not change. Thus, we can rewrite the algorithm in this equivalent (for the required output) form:

1. Find any longest subsequence of consecutive empty stalls.
2. Choose the middle or one of the two middle stalls.

Notice that even though there are still ties to be broken, the output is equivalent for all of them. Since the output is equivalent, so is the multiset of lengths of consecutive runs of empty stalls left behind, so the whole process only depends on that multiset. (As a reminder, a multiset is a set in which the same element can appear more than once.) We can write an optimized simulation that solves test set 2 following this pseudocode:

```
S = {N}  - This is a multiset!
repeat K times:
  X = max(S)
  X0 = ceil((X - 1) / 2)
  X1 = floor((X - 1) / 2)
  if this is the last step:
    we are done; answer is X0 and X1
  else:
    remove one instance of X from S
    insert X0 and X1 into S
```

If the operations over `S` are efficient, this will run in quasilinear time. There are many data structures that support insertion, finding the maximum, and removal of the maximum in logarithmic time, including AVL trees, red-black trees, and heaps. Many languages have one such structure in their standard libraries (e.g., the `multiset` or `priority_queue` in C++, `TreeSet` in Java, and `heapq` module in Python). Since we take O(log **K**) time for each of K steps, the algorithm takes only O(**K** log **K**) time, which is fast enough to solve test set 2. However, for test set 3, even quasilinear time on K is not enough.

Test set 3

The observation required to solve test set 3 is that we are simulating similar steps over and over again. The first time a bathroom user arrives, we partition **N** into ceil((**N** - 1) / 2) and floor((**N** - 1) / 2), which means that numbers between ceil((**N** - 1) / 2) and **N** will never appear in S. This hints at a logarithmic number of simulation steps.

Let's divide the work in stages. The first stage processes only **N**. Then, stage *i*+1 processes all of the values spawned by stage *i*. So, stage 2 processes up to 2 values: ceil((*i* - 1) / 2) and floor((*i* - 1) / 2). What about the other stages? It is not hard to prove by induction that they also process at most two consecutive values: since stage i processes two consecutive values, they are either 2x and 2x+1 or 2x and 2x-1, for some x (that is, one even and one odd number). Thus, the spawned values for stage i+1 can only be x and/or x-1. Since the largest value in each stage is at most half the largest value of the previous stage, there are a logarithmic number of stages. This all means that there are at most O(log **N**) different values that go into S at any point. Of course, some of them appear in S many, many times. So, the optimization to get the running time low enough for test set 3 is to process all repetitions of a given value at the same time, since all of them yield the same X0 and X1 values. We can do that by using a regular set with a separate count for the number of repetitions.

```
S = {N}  - This is a set, not a multiset!
C(N) = 1
P = 0
repeat:
  X = max(S)
  X0 = ceil((X - 1) / 2)
  X1 = floor((X - 1) / 2)
  P = P + C(X)
  if P ≥ K:
    we are done; the answer is X0 and X1.
  else:
    remove X from S
    insert X0 and X1 into S
    add C(X) to the counts of X0 and X1 in C
```

Once again, we have structures that implement all the required operations in logarithmic time, yielding an O(log^2 **N**) running time overall. In general, adding any good dictionary implementation to the structure of choice from the test set 2 solution would work, either by plugging the dictionary functionality into the structure (like `map` in C++ or `TreeMap` in Java) or having a separate hash-table for the dictionary (which is the easiest implementation in Python).

Moreover, since we proved the population of S is at most 4 at any given time (only values from two consecutive stages can coexist in S), any implementation of set and dictionary will provide all operations in constant time, because the size of the whole structure is bounded by a constant! This makes the overall time complexity just O(log **N**).

This was a nice problem to put experimentation to work if your intuition was not enough. After solving test set 1, if you print the succession of values for a fixed **N**, you may spot the pattern of few values occurring in the set S, and from there, you can find the mathematical arguments to support the needed generalization. In harder problems in later rounds, this can become an even more important asset to tackle problems. As you can see in many parts of last year's finals live stream, finalists use experimentation a lot to inspire themselves and/or validate their ideas before committing to them.