Code Jam 2008 - Round 1B

Analysis: Mousetrap

In this nice problem, there are two viewpoints one can start with.

- The current position is fixed and the array of cards is rotating each time.
- Think of the cards as a fixed ring (represented byan array) and the current position in focus as a pointer moving on the ring.

For most of us (and our programs), it is more convenient to adopt the second viewpoint.

After reading the story, it is not hard to see that the task is clear: put card 1 in the first position, then for each card i (in the order 2, 3, ..., **K**), we start from the current position, and find the i-th empty spot to the right, wrap around as many times as necessary, then put card i there.

The problem is to simulate the process described above. What makes it interesting is that **K** can be as big as 1000000, which makes the naive $\Theta(\mathbf{K}^2)$ simulation too slow for our 8-minute time limit. For each step, we need to compute the next position much faster than $\Theta(\mathbf{K})$ as in the naive approach. We describe three solutions below.

Solution A.

Let $\mathbf{S} = \sqrt{\mathbf{K}}$, we partition the **K** positions into **S** intervals of roughly equal size (also **S**). In addition to bookkeeping which positions are occupied (an array of size **K**, we call the first level counter), we also count for each interval how many positions are occupied (an array of size **S** that we call the second level counter). With this information, we may skip intervals of length **S** as many as possible, until we arrive at an interval where we know the card must belong to. Then in that interval we only need to deal with at most **S** first level counters.

Once we put down a card, it is a simple matter to update the counters. We only need to update one on the first level and one on the second level.

This solution runs in time $O(\mathbf{K}^{1.5})$.

Solution B.

Push the idea in the previous solution further. Why not have more levels of counters? In fact, one nice plan is to organize the levels into a binary tree. On the bottom (first) level of the tree we have each position as a separate interval. Every time we go up one level, we combine every other interval with the next one. Thus we will have log**K** levels; the top level being a single interval with all the positions. We omit the details, since we will see this again in the analysis of a Round 1C problem. We mention that the total number of counters is $O(\mathbf{K})$, and for each card we will need $O(\log \mathbf{K})$ time to find the position and another $O(\log \mathbf{K})$ time to update the counters. The running time of this method is $O(\mathbf{K} \log \mathbf{K})$.

For similar ideas in computer science, we refer to the Wiki page for interval trees.

Solution C.

Now let us do something different. At each step, after one position is occupied by card number i, we delete the position from the deck.

Notice that **n**, the number of queries is at most 100. We do not need to relabel all the positions, it is enough to do this for those **n** that we are interested in.

The solution can be implemented in two flavors, based on which viewpoint in the beginning of the analysis you pick. The short C++ program is, again, based on the second one, where the position (pos) changes as a pointer, and the deck does not move, except we delete one position in each step.

```
for (int j = 0; j < n; j++) answers[j] = -1;
for (int i = 1, pos = 0; i <= K; i++) {
  // Compute the next position, after wrap-around.
  pos = (pos + i - 1) % (K - i + 1);
  for (int j = 0; j < n; j++)
  if (answers[j] < 0) {
    if (queries[j] == pos+1) {
      queries[j] = -1; answers[j] = i;
    } else if (queries[j] > pos+1) {
      // The effect of deleting the next position.
      queries[j]--;
    }
  }
}
```

You can use a trick to combine the two arrays queries[] and answers[] into one. The programs runs in Θ(**n K**) time.

More information:

Interval trees