Editorial 6: Big Ben's Big Brain Bamboozles a Bovine 4+2+3=9 Points

Problem ID: `rotate`
Rank: `2+2+3`

Solution implementations for this problem can be found on our GitHub Repository for this contest!

Main Test Set

For the main test set, this problem can be pretty easily solved by just simulating the described process without any complicated data structures. Represent the deck using a list, and then simualte shuffling the deck $\mathbf{N}$ times (or $\mathbf{N}-1$ times, since the last shuffle moves the bottom card to the bottom, doing nothing).

Simulation

We can use a list of integers to represent the deck of cards being shuffled. Begin by inserting $1, \ldots, \mathbf{N}$ in ascending order. Let the integer at index 0 of the list represent the card at the top and let the integer at index $\mathbf{N}-1$ represent the card at the bottom.

To shuffle, loop the variable i over integers from 0 to $\mathbf{N}-1$. This represents the index of the card we move to the bottom. Each iteration, we delete the card at index i from the list, and then append it to the end of the list, and then increment i.

After the shuffle, simply index into the list with $\mathbf{K}-1$ to get the answer.

To implement this in each language:

- Python
  - Use the builtin `list`.
  - Use `l.pop(i)` and `l.append(i)`.
- Java
  - Use `java.util.ArrayList<Integer>`.
  - Use `l.remove(i)` and `l.add(i)`.
- C++
  - Use `std::vector<int>`.
  - Use `l.erase(l.begin() + i)` and `l.push_back(i)`.

Bonus Test Set 1

The time complexity of the simulation solution of the main test set is $O(\mathbf{N}^2)$. This is because removing an arbitrary value from the middle of a list takes linear time. With $1 \leq \mathbf{N} \leq 10^6$, this is too slow. To speed things up, we can make some observations and/or use more efficient data structures.

Queue

Observe that after shuffling up to index i, the values between 0 and i become locked in place, as the shuffling process will never change them ever again. As such, we can split our representation of the deck into two: we represent the values between $i+1$ and $\mathbf{N}-1$ using a queue, while using a regular list to represent the values between 0 and i.

A queue is a data structure that supports insertions at the front and deletions at the end of a sequence of elements. Most programming languages have a queue implementation in their standard library that is optimized for fast $O(1)$ insertion and deletion.

Begin by loading all the cards into the queue. To shuffle, remove a value from the front of the queue and insert it back into the end of the queue. This is equivalent to moving a card. Then, remove the next element of the queue and insert it to the end of the list. This is equivalent to incrementing our shuffle index. Repeat this process until the queue runs out.

The final list has our answer, so we simply index into it the same way we did for the main test set.

Since performing insertions and deletions from queues and lists for a single shuffle takes $O(1)$ time and we make $\mathbf{N}$ shuffles, the complexity of this algorithm is $O(\mathbf{N})$.

To implement this in each language:

- Python
  - Use `collections.deque()`.
  - Use `d.popleft()` and `d.append(card)`.
- Java
  - Use `java.util.ArrayDeque<Integer>`.
  - Use `d.removeFirst()` and `d.addLast(card)`.
- C++
  - Use `std::queue<int>`.
  - Use `d.pop()` and `d.pop(card)`.

Linked List

We can also solve this problem using a linked list. A linked list is a data structure that represents a list using a sequence of nodes. Each node contains a list value and also a pointer to the next node.

The idea is that we represent the deck as a linked list. Maintain pointers to the start of the list, the end of the list, and also the current index being shuffled. Each time we want to shuffle, we make the node before the current index point to the node after the current index. We also make the node at the end of the list point to the node at the current index.

To get the final answer, simply start at the start node and follow pointers forward $\mathbf{K}$ times. The value at that node has our final answer.

Since reassigning pointers for each shuffle takes $O(1)$ and we need to shuffle $\mathbf{N}$ times, the complexity of this algorithm is $O(\mathbf{N})$.

Deckless

We can even solve this problem without actually storing all the cards in the deck at all! To do this, we track the index j of the card that would end up in position $\mathbf{K}$, and then simulate the process in reverse. j begins as $\mathbf{K} - 1$. If a shuffle happens at an index i smaller than j, we increment j by 1 because the reverse of this process is inserting the last card into index i, pushing all the cards after it. If a shuffle happens when $j = \mathbf{N} - 1$, we change set j equal to i.

We didn't implement this solution ourselves. If you solved the problem this way, feel free to submit a pull request to add it to our repo!

Bonus Test Set 2

With $1 \leq \mathbf{N} \leq 10^{18}$, the deck is too big to even store all the values of the cards in memory. We need a clever solution that only tracks the relevant card that would end up in the $\mathbf{K}^{\text{th}}$ position without wasting time and memory on the cards that don't matter.

Divide and Conquer

Let's start by taking a look at the case when $\mathbf{N}$ is even.

Observe that after shuffling exactly $\frac{\mathbf{N}}{2}$ times, all of the even numbers will be locked into the first half in increasing order, and all of the odd numbers will be locked into the second half in increasing order.

For example, shuffling $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ 5 times will give us $[\mathbf{2}, \mathbf{4}, \mathbf{6}, \mathbf{8}, \mathbf{10}, 1, 3, 5, 7, 9]$ where the bolded numbers represent cards that have been locked in place.

If $\mathbf{K}$ is also even, the answer must be $\frac{\mathbf{K}}{2}$, so we can just return that.

Otherwise, if $\mathbf{K}$ is odd, we can recursively redefine the remaining shuffles! Relabel all of the unshuffled odd numbered cards by floor dividing the numbers by 2. For example, if we had $1, 3, 5, 7, 9$, we would relabel them to $1, 2, 3, 4, 5$. Since we relabeled all the cards, we need to relabel $\mathbf{K}$ too as $\frac{\mathbf{K}}{2} + 1$.

We know the answer must be somewhere in the remaining $\frac{\mathbf{N}}{2}$ unshuffled cards, so our answer must be $\frac{\mathbf{N}}{2}$ added to the result of recursively solving the subproblem with relabeled cards and $\mathbf{K}$ described above.

A similar case exists for when $\mathbf{N}$ is odd. The card labeled 1 ends up in an awkward position in the middle of the deck, but you can still account for it by treating $\mathbf{K} = 1$ as a separate case and then adjusting the relabelling and offset carefully for all other odd numbers.

The analysis is straightforward. In the worst case, $\mathbf{K}$ is always odd with each split. We then need to recurse with $\frac{\mathbf{N}}{2}$. $\mathbf{N}$ can only be split $\log_2(N)$ times before reaching the trivial case of where we only have one card. Within each recurrence, we only do simple arithmetic in $O(1)$. As such, the overall complexity of this algorithm is $O(\log(\mathbf{N}))$.

Design Notes

Conception

For the longest time, I wanted to write a divide and conquer problem that was accessible to beginner contestants. I personally found writing such a problem that was simultaneously interesting, simple, and engaging to an audience of broad skill levels to be quite a challenge. I went through many design iterations, but often they ended up too complicated, too hard, or otherwise just not very fun.

I was hanging out with some peeps in the CALICO community discord server playing CodinGame when inspiration struck! I was solving some lame implementation problem (that I unfortunately was unable to find the link to) when I realized some modifications could yield an interesting pattern. With a few more adjustments, `rotate` was created! Many people on the CALICO Team thought it was a really cool problem with an elegant solution.

Further Reading

In hindsight, the problem shares a number of overlapping properties with the related Josephus problem which itself is also very interesting. However, this wasn't intentional in the original design of the problem. The first chapter of Graham, Knuth, and Patashnik's Concrete Mathematics presents an excellent analysis and discussion of the Josephus problem and related generalizations that I highly recommend checking out if you thought this problem was cool!

Problem Credits

- Idea: alfphaderp
- Problem Statement: alfphaderp, Kaashvi, 1 more anonymous author
- Art: Kaashvi
- Implementation: alfphaderp, JJ, Chris
- Testing: Ariel Shehter, yjp20, qxbytes
- Editorial: alfphaderp