Introduction

In looking at merge sort a fundamental operation that was used was merging two halves of an array into a single one based on the order of the relevant elements. What if we do that merge randomly instead? Then we're shuffling (more specifically, performing a *riffle* shuffle).

In this lab you'll implement a simple version of that shuffle and investigate how well it works.

Problem description

Your code `shuffle(int[] d)` is to do the following:

- Split `d` into two halves (or near-halves if it has odd length)
- Merge the two halves back into `d` choosing randomly based on a coin-flip each time to choose which half to merge from (so long as cards are available in both halves).

It should be obvious that a single iteration will not result in a randomised deck. The idea will then be to begin with a sorted deck `d` (where `d[i] = i`), repeatedly shuffle it, and gather information about where the 0 is. How many shuffles are needed before this is reasonably evenly distributed through the deck?

Problems

- Write the `shuffle(int[] d)` (shuffle once) method.
- Use it for a `shuffle(int[] d, int k)` method that does k shuffles.
- Collect data for various k using a starting deck of size 52 with `d[i] = i` about where card 0 winds up after k shuffles.
- If the shuffling is effective, there should be little or no bias in the final position of 0, i.e., all positions should be equally likely. How large does k need to be before this seems to be (approximately) correct?
- What would happen if we used no randomness, i.e., just alternately picked cards from each half?
- How should we *really* shuffle an array so that, in one call to a method like `shuffle`, we obtain every possible rearrangement of the cards with the same likelihood?