AP ACTIVITY **4.2.3**

Create Shuffle Algorithms (AP)

INTRODUCTION

What makes a game different from an exercise or a story? Some might suggest the random behavior or the element of surprise is inherent in most games. Random behavior, such as a surprise event or an unknown amount of time passing, makes a game interesting. If you always use an ordered deck of cards, a card game would have identical outcomes each time you played; if you have ever played a card game, you know you have to shuffle the deck before playing the game.

Materials

- Computer with BlueJ IDE

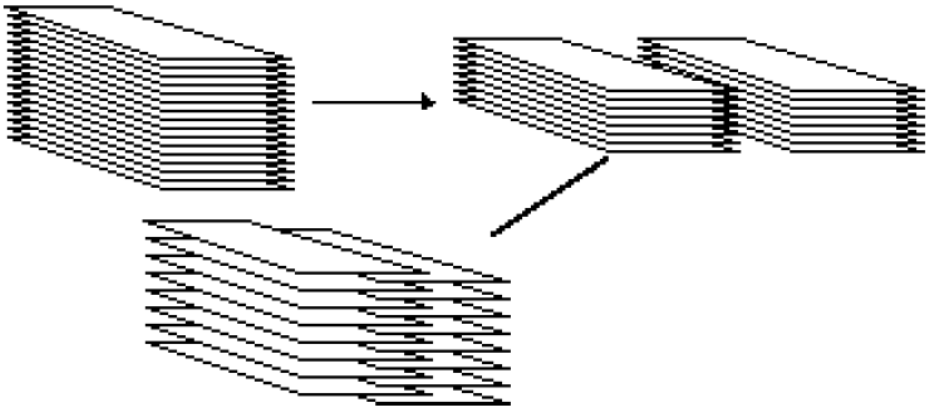
Procedure

You will now explore the shuffling of a deck, that is, the permutation of its cards into a random-looking sequence. A requirement of the shuffling procedure is that any particular permutation has just as much chance of occurring as any other. You will use the `Math.random` method to generate random numbers to produce these permutations.

Several ideas for designing a shuffling method come to mind. Consider the following two methods:

Perfect Shuffle

Card players often shuffle by splitting the deck in half and then interleaving the two half-decks, as shown:



This procedure is called a perfect shuffle if the interleaving alternates between the two half-decks. Unfortunately, the perfect shuffle comes nowhere near generating all possible deck permutations. In fact, eight shuffles of a 52-card deck return the deck to its original state!

Consider the following "perfect shuffle" algorithm, written in pseudo-code, that starts with an array named `cards`. The array contains 52 cards and algorithm creates new array named `shuffled`.

Initialize `shuffled` to contain 52 "empty" elements
Set `k` to 0
For `j` = 0 to 25,
  – Copy `cards[j]` to `shuffled[k]`
  – Set `k` to `k+2`
Set `k` to 1
For `j` = 26 to 51,
  – Copy `cards[j]` to `shuffled[k]`
  – Set `k` to `k+2`

This approach moves the first half of `cards` to the even index positions of `shuffled`, and it moves the second half of `cards` to the odd index positions of `shuffled`.

The above algorithm shuffles 52 cards. If an odd number of cards is shuffled, the array `shuffled` has one more even-indexed position than odd-indexed positions. Therefore, the first loop must copy one more card than the second loop does. This requires rounding up when calculating the index of the middle of the deck. In other words, in the first loop j must go up to `(cards.length + 1)/2`, exclusive, and in the second loop `j` must begin at `(cards.length + 1)/2`.

Selection Shuffle

Consider the following algorithm that starts with an array named `cards` that contains 52 cards and creates an array named `shuffled`. We will call this algorithm the "selection shuffle."

Initialize `shuffled` to contain 52 "empty" elements.
Then for `k` = 0 to 51,
- Repeatedly generate a random integer `j` between 0 and 51, inclusive until `cards[j]` contains a card (not marked as empty)
- Copy `cards[j]` to `shuffled[k]`
- Set `cards[j]` to empty

This approach finds a suitable card for the kth position of the deck. Unsuitable candidates are any cards that have already been placed in the deck.

While this is a more promising approach than the perfect shuffle, its big defect is that it runs too slowly. Every time an empty element is selected, it has to loop again. To determine the last element of `shuffled` requires an average of 52 calls to the random number generator.

A better version, the "efficient selection shuffle," works as follows:

For `k` = 51 down to 1,
- Generate a random integer `r` between 0 and `k`, inclusive
- Exchange `cards[k]` and `cards[r]`

This has the same structure as selection *sort*:

For `k` = 51 down to 1,
- Find `r`, the position of the largest value among `cards[0]` through `cards[k]`
- Exchange `cards[k]` and `cards[r]`

The selection shuffle algorithm does not require a loop to find the largest (or smallest) value to swap, so it works quickly.

1. Open and create a BlueJ project for *ElevensActivity3* with the source files from your teacher. Use file `Shuffler.java` file to implement the perfect shuffle and the efficient selection shuffle methods as described above. Note that you will be shuffling arrays of integers.
2. `Shuffler.java` also provides a `main` method that calls the shuffling methods. Execute the `main` method and inspect the output to see how well each shuffle method actually randomizes the array elements. You should execute main with different values of `SHUFFLE_COUNT` and `VALUE_COUNT`.

CONCLUSION

1. Write a static method named `flip` that simulates a flip of a weighted coin by returning either "heads" or "tails" each time it is called. The coin is twice as likely to turn up heads as tails. Thus, flip should return "heads" about twice as often as it returns "tails."
2. Write a static method named `arePermutations` that, given two `int` arrays of the same length but with no duplicate elements, returns `true` if one array is a permutation of the other (i.e., the arrays differ only in how their contents are arranged). Otherwise, it should return `false`.
3. Suppose that the initial contents of the `values` array in `Shuffler.java` are `{1, 2, 3, 4}`. For what sequence of random integers would the efficient selection shuffle change values to contain `{4, 3, 2, 1}`?