Write a function for doing an in-place↴ shuffle of an array.

The shuffle must be "uniform," meaning each item in the original array must have the same probability of ending up in each spot in the final array.

Assume that you have a function `get_random(floor, ceiling)` for getting a random integer that is >= `floor` and <= `ceiling`.

Gotchas

A common first idea is to walk through the array and swap each element with a random other element. Like so:

```
def get_random(floor, ceiling)
    rand(floor..ceiling)
end

def naive_shuffle(the_array)

    # for each index in the array
    (0...the_array.length).each do |first_index|

        # grab a random other index
        second_index = get_random(0, the_array.length - 1)

        # and swap the values
        if second_index != first_index
            the_array[first_index], the_array[second_index] = the_array[second_index], the_array
        end
    end
end
```

However, this does not give a uniform random distribution.

Why? We could calculate the exact probabilities of two outcomes to show they aren't the same. But the math gets a little messy. Instead, think of it this way:

Suppose our array had 3 elements: [a,b,c]. This means it'll make 3 calls to `get_random(0, 2)`. That's 3 random choices, each with 3 possibilities. So our total number of possible *sets of choices* is $3 * 3 * 3 = 27$. Each of these 27 sets of choices is equally probable.

But how many possible *outcomes* do we have? If you paid attention in stats class you might know the answer is 3!, which is 6. Or you can just list them by hand and count:

```
a,b,c
a,c,b
b,a,c
b,c,a
c,b,a
c,a,b
```

But our function has 27 equally-probable sets of choices. 27 is not evenly divisible by 6. So some of our 6 possible *outcomes* will be achievable with more *sets of choices* than others.

We can do this in a single pass. $O(n)$ time and $O(1)$ space.

A common mistake is to have a mostly-uniform shuffle where an item is less likely to stay where it started than it is to end up in any given slot. Each item should have the same probability of ending up in each spot, including the spot where it starts.

Breakdown

It helps to start by ignoring the " in-place↴ " requirement, then adapt the approach to work in-place.

Also, the name "shuffle" can be slightly misleading—the point is to arrive at a random ordering of the items from the original array. Don't fixate too much on preconceived notions of how you would "shuffle" e.g. a deck of cards.

How might we do this by hand?

We can simply choose a random item to be the first item in the resulting array, then choose another random item (from the items remaining) to be the second item in the resulting array, etc.

Assuming these choices were in fact random, this would give us a uniform shuffle. To prove it rigorously, we can show any given item a has the same probability ($\frac{1}{n}$) of ending up in any given spot.

First, some stats review: to get the probability of an outcome, you need to *multiply the probabilities of all the steps required for that outcome*. Like so:

Outcome	Steps	Probability
item #1 is a	a is picked first	$\boldsymbol{\frac{1}{n}}$
item #2 is a	a not picked first, a picked second	$\frac{(n-1)}{n} * \frac{1}{(n-1)} = \boldsymbol{\frac{1}{n}}$
item #3 is a	a not picked first, a not picked second, a picked third	$\frac{(n-1)}{n} * \frac{(n-2)}{(n-1)} * \frac{1}{(n-2)} = \boldsymbol{\frac{1}{n}}$
item #4 is a	a not picked first, a not picked second, a not picked third, a picked fourth	$\frac{(n-1)}{n} * \frac{(n-2)}{(n-1)} * \frac{(n-3)}{(n-2)} * \frac{1}{(n-3)} = \boldsymbol{\frac{1}{n}}$

So, how do we implement this in code?

If we didn't have the "in place" requirement, we could allocate a new array, then one-by one take a random item from the input array, remove it, put it in the first position in the new array, and keep going until the input array is empty (well, probably a *copy* of the input array—best not to destroy the input)

How can we adapt this to be in-place?

What if we make our new "random" array simply be the *front* of our input array?

Solution

We choose a random item to move to the first index, then we choose a random *other* item to move to the second index, etc. We "place" an item in an index by swapping it with the item currently at that index.

Crucially, once an item is placed at an index it can't be moved. So for the first index we choose from n items, for the second index we choose from $n - 1$ items, etc.

```ruby
def get_random(floor, ceiling)
    rand(floor..ceiling)
end

def shuffle(the_array)
    # if it's 1 or 0 items, just return
    if the_array.length <= 1
        return the_array
    end

    last_index_in_the_array = the_array.length - 1

    # walk through from beginning to end
    (0..the_array.length - 2).each do |index_we_are_choosing_for|

        # choose a random not-yet-placed item to place there
        # (could also be the item currently in that spot)
        # must be an item AFTER the current item, because the stuff
        # before has all already been placed
        random_choice_index = get_random(index_we_are_choosing_for, last_index_in_the_array

        # place our random choice in the spot by swapping
        if random_choice_index != index_we_are_choosing_for
            the_array[index_we_are_choosing_for], the_array[random_choice_index] =
                the_array[random_choice_index], the_array[index_we_are_choosing_for]
        end
    end
end
```

This is a semi-famous algorithm known as the **Fisher-Yates shuffle** (sometimes called the Knuth shuffle).

Complexity

$O(n)$ time and $O(1)$ space.

What We Learned

Don't worry, most interviewers won't expect a candidate to know the Fisher-Yates shuffle algorithm. Instead, they'll be looking for the problem-solving skills to *derive* the algorithm, perhaps with a couple hints along the way.

They may also be looking for an understanding of why the naive solution is non-uniform (some outcomes are more likely than others). If you had trouble with that part, try walking through it again.