Randomness

Objectives

- Learn the difference between randomness and pseudorandomness
- Learn how how to shuffle an array in O(n) time and space complexity

What is randomness?

What is Randomness? This seems like a simple question but randomness is actually one of the most commonly misunderstood concepts in math and computing. Here's a definition of randomness from wikipedia:

```
Randomness is the lack of pattern or predictability in events. A random sequence
of events, symbols or steps has no order and does not follow an intelligible
pattern or combination.
```

Coin flips are random: If you flip a coin 9 times and get 9 **heads** in a row, the probability of hitting **heads** a 10th time is still exactly 50%.

Rock-Paper-Scissors is NOT random when played with humans. It turns out, people are quite poor at randomizing which makes it possible to gain an edge over opponents using psychological analysis.

Try this exercise:

```
Write down 10 random numbers between 1-10.
```

What were you thinking while choosing numbers? Did you have thoughts like, "I already picked 7 twice so I shouldn't pick 7 again," or, "I need a 5 since there hasn't been one yet." Everytime you use rationale like this, you add personal bias, making your choices more predictable and less random.

Computers are pretty good at this sort of randomness. Here's a Python list comprehension that runs this exercise 10 times in a row:

```
>>> import random
>>> [random.randint(1,10) for i in range(0,10)]
[6, 2, 2, 6, 6, 1, 7, 9, 9, 4]
>>> [random.randint(1,10) for i in range(0,10)]
[1, 3, 2, 8, 9, 3, 1, 9, 4, 8]
>>> [random.randint(1,10) for i in range(0,10)]
[6, 4, 9, 4, 5, 5, 4, 7, 3, 7]
>>> [random.randint(1,10) for i in range(0,10)]
[6, 6, 1, 1, 2, 8, 9, 2, 3, 10]
>>> [random.randint(1,10) for i in range(0,10)]
[8, 1, 9, 5, 10, 4, 6, 2, 10, 7]
```

```
>>> [random.randint(1,10) for i in range(0,10)]
[8, 4, 1, 1, 8, 6, 9, 4, 1, 9]
>>> [random.randint(1,10) for i in range(0,10)]
[2, 1, 2, 2, 2, 10, 1, 1, 1, 3]
>>> [random.randint(1,10) for i in range(0,10)]
[1, 6, 1, 6, 6, 4, 9, 9, 7, 5]
>>> [random.randint(1,10) for i in range(0,10)]
[4, 4, 7, 6, 6, 5, 2, 8, 3, 7]
>>> [random.randint(1,10) for i in range(0,10)]
[8, 1, 4, 3, 4, 3, 5, 4, 4, 9]
```

Looking at these results, you may think that the lists seem LESS random. After all, the first list contains three 6s, two 2s, two 9s, and no 3s, 5s, 8s or 10s. The seventh list is 80% 1s and 2s. Is this REALLY random?

The short answer is **yes**. Small random sample sizes will show statistical variation. The larger the sample size though, the closer the result will skew toward the expected values. This is known as the Law of Large Numbers. Consider the following function:

```
import random
def random_counts(n):
    """
    Generates n random numbers between 1-10 and counts how many of each there are.
    """
    counts = {1:0,2:0,3:0,4:0,5:0,6:0,7:0,8:0,9:0,10:0}
    nums = [random.randint(1,10) for i in range(1,n)]
    for i in nums:
        counts[i] += 1
    return counts
```

This will generate a list of n random numbers then count how many of each there are. Let's try running it for different values of n.

```
>>> random_counts(10)
{1: 0, 2: 2, 3: 0, 4: 1, 5: 2, 6: 2, 7: 2, 8: 0, 9: 0, 10: 0}
>>> random_counts(10)
{1: 2, 2: 0, 3: 3, 4: 0, 5: 2, 6: 1, 7: 0, 8: 0, 9: 1, 10: 0}
>>> random_counts(10)
{1: 1, 2: 1, 3: 1, 4: 0, 5: 0, 6: 0, 7: 2, 8: 1, 9: 2, 10: 1}
>>> random_counts(100)
{1: 9, 2: 6, 3: 10, 4: 14, 5: 14, 6: 8, 7: 7, 8: 11, 9: 10, 10: 10}
>>> random_counts(100)
{1: 6, 2: 7, 3: 13, 4: 11, 5: 16, 6: 9, 7: 5, 8: 13, 9: 8, 10: 11}
>>> random_counts(100)
{1: 11, 2: 11, 3: 8, 4: 10, 5: 12, 6: 7, 7: 8, 8: 13, 9: 10, 10: 9}
>>> random_counts(1000)
{1: 110, 2: 106, 3: 98, 4: 96, 5: 98, 6: 96, 7: 90, 8: 98, 9: 111, 10: 96}
>>> random_counts(1000)
{1: 102, 2: 87, 3: 103, 4: 101, 5: 95, 6: 100, 7: 107, 8: 94, 9: 94, 10: 116}
>>> random_counts(1000)
```

```
{1: 107, 2: 106, 3: 119, 4: 95, 5: 99, 6: 96, 7: 94, 8: 92, 9: 91, 10: 100}
>>> random_counts(10000)
{1: 971, 2: 1015, 3: 1024, 4: 1004, 5: 956, 6: 952, 7: 1054, 8: 978, 9: 1019, 10:
1026}
>>> random_counts(10000)
{1: 1024, 2: 996, 3: 970, 4: 1029, 5: 1014, 6: 997, 7: 977, 8: 936, 9: 1030, 10:
1026}
>>> random_counts(10000)
{1: 1086, 2: 968, 3: 1010, 4: 1019, 5: 973, 6: 966, 7: 1004, 8: 1006, 9: 996, 10:
971}
>>> random_counts(1000000)
{1: 100270, 2: 100035, 3: 100127, 4: 99814, 5: 99748, 6: 100162, 7: 99851, 8:
99706, 9: 100104, 10: 100182}
>>> random_counts(1000000)
{1: 100784, 2: 100029, 3: 100287, 4: 99821, 5: 99933, 6: 100350, 7: 100419, 8:
99304, 9: 99329, 10: 99743}
>>> random_counts(1000000)
{1: 100170, 2: 100205, 3: 99644, 4: 99702, 5: 99874, 6: 100316, 7: 99996, 8:
99864, 9: 99968, 10: 100260}
```

Statistically, you would expect each number to show up exactly 10% of the time. As you can see, the larger the n, the closer the results get to that expected percentage. This is how casinos work: they may lose a large amount of money on a single roll of the roulette wheel but over thousands of spins, the house always comes out ahead in the long run.

Computers are quite good at producing statistical randomness like this. But is it **actually** random? Turns out, the answer is no.

What is pseudorandomness?

Computers are machines that take some input, run a set of operations on that input, then return an output. These operations are generally **deterministic**, which means it will always produce the same output from an identical set of inputs.

You can think of computers like powerful calculators: inputting `5+5` will always return 10 and `123 * 456` will always return 56088 but there's no way to generate a truly random value.

While computers are physically incapable of generating random numbers, they are quite adept at generating **pseudorandom** numbers. This involves taking an arbitrary input value called a **seed** and scrambling it with a deterministic algorithm. Let's take a look at this in action in Python.

```
>>> import random
>>> random.random()
0.5563574873081953
>>> random.random()
0.033120716851841814
>>> random.random()
0.0639699823278671
>>> random.random()
0.9781604581128546
```

```
>>> random.random()
0.47905656332349167
```

The `random.random()` function returns a random value between 0 and 1. Now, let's see what happens when we pass in a seed.

```
>>> random.seed(123)
>>> random.random()
0.052363598850944326
>>> random.random()
0.08718667752263232
>>> random.random()
0.4072417636703983
>>> random.random()
0.10770023493843905
>>> random.random()
0.9011988779516946
```

Nothing out of the ordinary. Now, let's reset the seed and try again:

```
>>> random.seed(100 + 20 + 3)
>>> random.random()
0.052363598850944326
>>> random.random()
0.08718667752263232
>>> random.random()
0.4072417636703983
>>> random.random()
0.10770023493843905
>>> random.random()
0.9011988779516946
```

As you can see, we get the exact same values! This is useful if you want to get a predictable chain of "random" numbers, like if you wanted to recreate a particular Minecraft map. With pseudorandom number generators, an entire world can be contained in one simple integer.

Shuffling an Array

Now that we know how computers generate random numbers, let's try shuffling an array. We can do this quite efficiently using an algorithm called the Fisher-Yates shuffle.

Here's the basic algorithm:

1. Begin with a list of N elements
2. Swap the first element with the i-th element, where i is a random position after the first.
3. Repeat this for each element, in order, until you reach the end of the list.

Let's try implementing the Fisher-Yates using the Python function, `random.randint(a, b)` which returns a random integer between a and b (including a and b). We'll start with a sorted list of N elements, where N=5.

```
>>> l = [0, 1, 2, 3, 4]
```

Now, we find a random index between 0 and N-1 (i.e. 4) to swap our first element with.

```
>>> random.randint(0, 4)
2
```

This returned `2`, so let's swap the first element with the item in index 2.

```
>>> swap = l[2]
>>> l[2] = l[0]
>>> l[0] = swap
>>> l
[2, 1, 0, 3, 4]
```

Notice that we needed to store l[2] in a temporary variable before swapping so it didn't get overwritten by l[0].

Let's move on to the next element:

```
>>> random.randint(1, 5)
4
```

Our random index returned `4`, so let's swap the second value with index 4.

```
>>> swap = l[4]
>>> l[4] = l[1]
>>> l[1] = swap
>>> l
[2, 4, 0, 3, 1]
```

Onto the third element!

```
>>> random.randint(2, 4)
2
```

Here, our random number generator returned `2`, which is the same index we want to swap, so we swap the number with itself. This is perfectly fine.

```
>>> swap = l[2]
>>> l[2] = l[2]
>>> l[2] = swap
>>> l
[2, 4, 0, 3, 1]
```

Let's finish up.

```
>>> random.randint(3, 4)
4
>>> swap = l[4]
>>> l[4] = l[3]
>>> l[3] = swap
>>> l
[2, 4, 0, 1, 3]
```

And we're done! Our list, `l` is now shuffled. We don't need to swap the last element since `random.randint(4,4)` will always return 4. We visited each element once and both finding a random index and swapping happens in O(1) time, so our overall shuffle occurs in O(n) time. We don't require any extra memory besides the swap variable, so our space complexity is O(n + 1), or just O(n) since we ignore the lesser factors.

We can generalize this into an algorithm like so:

```
import random
def fisher_yates_shuffle(l):
    for i in range(0, len(l) - 2):
        random_index = random.randint(i, len(l) - 1)
        swap = l[random_index]
        l[random_index] = l[i]
        l[i] = swap
```