Exercises

Note: you may not use any external packages apart from `math/rand`.

Relatively Prime Probability

Two numbers are called **relatively prime** if they do not share any factors (i.e., their GCD is equal to 1). A natural question is to determine the probability that two randomly selected integers are relatively prime. If we did not see a way of calculating this probability mathematically, then we could attempt to estimate the probability by generating every pair of integers in some range and counting the number of relatively prime pairs. For example, we could generate all pairs of numbers between 1 trillion and 2 trillion. However, there are about $5 \cdot 10^{23}$ pairs of integers in this range, so this approach would be computationally intensive.

One use of Monte Carlo simulation is to estimate a probability such as this one that could be computed via brute force but that would take an enormous amount of computational power.

Exercise: Write and implement a function **RELATIVELYPRIMEPROBABILITY** that takes three integers *x*, *y*, and *numPairs* as input, and returns an estimate of the probability that two randomly chosen numbers between *x* and *y* are relatively prime by selecting *numPairs* pairs of numbers between *x* and *y*, inclusively.

Then, call your function on a very wide range of values with *numPairs* equal to 1 million. What is your conjectured value for the probability that two integers are relatively prime?

The Birthday Problem

When we generate pseudorandom numbers, it is natural to think that a PRNG that produces repeated numbers is bad. After all, the brain's method for generating "random" numbers tends to avoid repeats. But as we will see, repetition is a natural component of randomness.

We can represent a sequence of random integers generated by a PRNG using an array. The following exercise will help us detect whether a sequence of integers has a repeated element.

Exercise: Write and implement a function **HASREPEAT** that takes an array of integers as input; this function should return "true" if there is a repeated value and "false" otherwise.

You may find **HASREPEAT** useful when writing the following function to explore repeated elements in sequences of random numbers.

Exercise: Imagine a room of *numPeople* people, each with a birthday on one of the 365 days of the year (February 29$^{\text{th}}$ babies are forbidden).

Write a function **BIRTHDAYPARADOX** that takes two integers, *numPeople* and *numTrials*. It runs *numTrials* simulations and returns the average number of trials for which in a room of *numPeople* randomly generated people, at least one pair of people have the same birthday. (Hint: the month is irrelevant when running these simulations.)

What is the smallest value of *numPeople* for which there is a greater than 50% chance of two people sharing the same birthday? Are you surprised?

The Middle-Square PRNG

Perhaps the Middle-Square approach is generally a good PRNG, and the poor-performing seeds that we saw for it are just outliers. More generally, we would like to know how we can we test the quality of a PRNG; in this case, we want to know how many seeds cause short "cycles" of numbers.

For a given seed, we will use the term **period** to refer to the length of the cycle of numbers generated by a PRNG with respect to a given seed. Our goal is to implement the Middle-Square approach and then determine how many seeds there are with short periods.

We have already determined whether or not a sequence of numbers has a repeat with **HASREPEAT**. If a sequence has a repeat, we would like to compute the period of the cycle generated by these numbers.

Exercise: Write a function **COMPUTEPERIODLENGTH()** that takes an array of integers as input. If there are no values that repeat, this function should return 0; if there is a repeated value in the input sequence, then the function should return the period of this sequence. Hint: if the sequence of numbers was $(1473, 2856, 9830, 1789, 4468, 9830)$, then the length of the period would be 3 because the numbers that will repeat are 9830, 1789, and 4468.

Next, we would like to write a function **SQUAREMIDDLE** that takes integers x and *numDigits* and returns the result of squaring x and taking its middle *numDigits* digits. This is a simple task by hand, but how can we teach it to a computer?

Exercise: We will need to ensure that *numDigits* is even. Write and implement a function **COUNTNUMDIGITS()** that takes an integer x as input and returns the number of digits in x. Your function should work for both positive and negative numbers. (Hint: first write your function for positive values of x, and then make a slight adjustment to it to accommodate negative values of x.)

We can divide the process of taking its middle *numDigits* digits of an integer into two steps: "crossing off" the first `numDigits/2` digits, then crossing off the last `numDigits/2` digits. For example, we would convert the number $x = 12345678$ into 345678, and then 3456. Note that the first process corresponds to taking the *remainder* when we divide x by $10^6 = 10^{8-2}$, and the second process corresponds to taking the *integer division* of the resulting number by 10^2. In general, we can now see that we are taking the remainder of x by $10^{2 \cdot numDigits - numDigits/2}$, and then dividing the re-

sulting number by $10^{numDigits/2}$. We only need to generalize this idea for an arbitrary value of *numDigits* to write **SQUAREMIDDLE**.

Exercise: Write and implement **SQUAREMIDDLE**. You should make sure to provide panic statements ensuring that *numDigits* is even, that both input parameters are positive, and that the number of digits in x is not greater than twice the value of *numDigits*. You will also find it helpful to write a subroutine **POW10** that takes an integer n and returns 10^n.

We can now use **SQUAREMIDDLE** as a subroutine in the following function, which generates the sequence of random numbers for the middle-square sequence corresponding to a given seed and number of digits.

```
GENERATEMIDDLESQUARESEQUENCE(seed, numDigits)
    seq ← sequence consisting of seed
    while HASREPEAT(seq) is false
        append SQUAREMIDDLE(seed, numDigits) to seq
    return seq
```

Exercise: Implement **GENERATEMIDDLESQUARESEQUENCE()**; test that it works on the example seeds 1600 and 3792 when *numDigits* is equal to 4. Then call **GENERATEMIDDLESQUARESEQUENCE()** on every four-digit seed between 1 and 9999. How many seeds produce a sequence of period 10 or smaller? Is the Middle-Square approach a good PRNG?