Exercises Part 0: Set up

Create a directory under `go/src` called `hw1`. Download the `functions.go` file provided online, and place it in hw1. We suggest testing each function **SOMEFUNCTION** that you write by calling `fmt.Println(SomeFunction(input))` within `func main()`, where `input` is suitable input. For example, use

```
fmt.Println(SumOfFirstIntegers(5))
```

to ensure your function is correctly returning 15.

When you submit your file to Autolab, remove `func main()` as well as `"fmt"` from your import packages. You will most likely retain the import of the `"math"` package due to the helpful `"math.Sqrt()"` function.

Exercises Part 1: Warming Up

In what follows, "write and implement" means to write the function in pseudocode first, and then implement your function in Go.

Exercise: Write and implement two versions of a function **SUMOFFIRSTNINTEGERS**, one with a for loop and one with a while loop. The function should take as input an integer n and return the sum $1+2+\cdots+n$. For example, **SUMOFFIRSTNINTEGERS**$(5) = 15$ and **SUMOFFIRSTNINTEGERS**$(10) = 55$.

Combinations, permutations, and a short guide to debugging

The number of ways to order n objects is $n!$, since we have n choices for the object in position 1, $n-1$ choices for the object in position 2, and so on. If we only want to order k of the objects, we still have n choices for the object in position 1, $n-1$, choices for the object in position 2, but this stops when we have the object in position k, where we have $n-k+1$ choices. This is called the **permutation statistic** $P(n,k)$ and is equal to the product $n \cdot (n-1) \cdots (n-k+1)$. Note that this expression can also be written using factorials as $n!/((n-k)!)$.

If we don't want to order the k objects, but instead are just selecting a subset of k from n objects, then we obtain the **combination statistic**, denoted $C(n,k)$. There are $k!$ different permutations corresponding to the same selection of objects, and so $C(n,k) = P(n,k)/k!$. In other words, $C(n,k) = n!/((n-k)! \cdot k!)$.

Exercise: Write and implement functions **COMBINATION**(n,k) and **PERMUTATION**(n,k) computing the combination and permutation statistics.

Exercise: Modify your implementations of **COMBINATION** and **PERMUTATION** so that they are able to compute **PERMUTATION**$(1000,2)$, **COMBINATION**$(1000,2)$, and **COMBINATION**$(1000,998)$ without any problem.

Arrays

Exercise: Implement **FACTORIALARRAY**.

Exercise: Implement **FIBONACCIARRAY**.

Exercise: Write and implement a function **MINARRAY** that takes an array of integers as input and returns the minimum of all these integers.

Exercise: Write and implement a function **GCDARRAY** that takes an array of integers as input and generalizes the idea in **TRIVIALGCD** to return the greatest common divisor of all of the integers in the array.

Implementing and timing prime finding algorithms

Exercise: Implement **TRIVIALPRIMEFINDER**.

Exercise: Implement **SIEVEOFERATOSTHENES**.

Exercise: Write and implement a function **LISTOFPRIMES** that takes an integer n and returns an array containing all of the primes up that are less than or equal to n. (Hint: use **SIEVEOFERATOSTHENES** as a subroutine.)

Time your implementations of the two prime-finding algorithms for input n equal to every power of 10 between a thousand and a billion. What do you observe?

Exercises Part 2: More Number Types, and Mathematical Conjectures

Perfect numbers

Although the Greeks were interested in prime numbers, they were just as intrigued by a collection of numbers that you might not have heard of. A **perfect number** is an integer n that is equal to the sum of its "proper" divisors (meaning those smaller than n). For example, 6 is perfect because $1+2+3=6$, and 28 is perfect because $1+2+4+7+14=28$. Perfect numbers are far rarer than prime numbers; the Greeks only knew of these two as well as 496 and 8928, and only 49 perfect numbers have ever been discovered. Surely if we know of so few perfect numbers, they must be finite?

Exercise: Write and implement a function **ISPERFECT** that takes an integer n as input and returns "true" if n is perfect and "false" otherwise. Recalling our first attempt at **ISPRIME**, do you see any ways of making your function more efficient?

The first four perfect numbers, and in fact all 49 known perfect numbers, have something in common: they're all even! No one has ever found an odd perfect number, and yet no one has ever been able to prove that odd perfect numbers don't exist.

The plot thickens when we notice the following pattern.

$$6 = 2^1(2^2 - 1)$$
$$28 = 2^2(2^3 - 1)$$
$$496 = 2^4(2^5 - 1)$$
$$8128 = 2^6(2^7 - 1)$$

Exercise: Write and implement a function **NEXTPERFECTNUMBER** that takes an integer n as input and uses **ISPERFECT** as a subroutine to find the smallest perfect number that is larger than n. Then use your function to find the fifth perfect number (which was unknown to the Greeks). Can this number be represented as a product of the above form?

The ancient Greeks knew that numbers of the form $2^m - 1$ are more likely than others to be prime; such numbers are called **Mersenne primes**. When $m = 4$, $2^m - 1$ is 15, which is certainly not prime. But $2^2 - 1 = 3$, $2^3 - 1 = 7$, $2^5 - 1 = 31$, and $2^7 - 1 = 127$ are all prime.

In 1876, Edouard Lucas would demonstrate that $2^{127} - 1$ was prime after *19 years* of manual computations! The rise of computers has made Lucas's work seem particularly lonely — in 2017, a computer program discovered the primality of $2^{77232917} - 1$, a number so large that it has 23 million *digits*.

The connection between two apparently very different types of numbers is perplexing and leads us to a conjecture. Could it be that every perfect number is formed from a Mersenne prime?

The **Euclid-Euler Theorem** confirms our intuition and states that every *even* perfect number must be of the form $2^{m-1} \cdot (2^m - 1)$, and vice-versa: if $2^m - 1$ is a Mersenne prime, then the number $2^{m-1} \cdot (2^m - 1)$ must be perfect. As a result of the Euclid-Euler Theorem, we know that if there are infinitely many Mersenne primes, then there are infinitely many perfect numbers. However, no one knows if there are infinitely many Mersenne primes, and so the infinitude of the perfect numbers remains unknown.

Twin primes

Twin primes are pairs of prime numbers that are only 2 apart (such as 3 and 5, or 29 and 31). Much like perfect and amicable numbers, no one knows if there are infinitely many pairs of twin primes, although computations have indicated that they do go on forever, leading to the **Twin Primes Conjecture** that there are infinitely many such pairs.

Exercise: Write and implement a function **NEXTTWINPRIMES** that takes an integer n as input and returns the smallest pair of twin primes that are both larger than n, using **ISPRIME** as a subroutine.

The Hailstone sequence and the Collatz conjecture

The **Hailstone function** $h(n)$ is defined by:

$$h(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases}$$

The **Hailstone sequence** for n is defined by repeatedly applying this function until it reaches 1. For example, the Hailstone sequence for $n = 19$ is

$$19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.$$

Exercise: Write a function **HAILSTONESEQUENCE** that takes an integer n and returns an array containing the Hailstone sequence of n. As always, it will be helpful to use a subroutine. Then use your function to find the number smaller than 1 million that has the longest hailstone sequence.

It has been conjectured (but never proven!) that for all n, the Hailstone sequence does reach 1. This statement is called the **Collatz Conjecture**[3]. Despite the Collatz conjecture being known for nearly 100 years, there has been little progress on proving it, with 20th Century mathematician Paul Erdős famously lamenting,

> *Mathematics is not yet ripe enough for such questions.*

[3] There is an excellent short video on the Collatz Conjecture at `http://bit.ly/2aHh0vp`