4.5 Justifying Correctness (Beyond Using Test Cases)

While writing test cases—either unit tests or property-based tests—is a useful way to give us confidence that our code is correct, they cannot *guarantee* correctness. Even our discussion of property-based tests for `is_even` at the end of the previous section runs into the fundamental limitation that a computer can only check properties on a finite number of inputs, and so we can't "know for sure" that the property will hold for values that weren't checked. Fundamentally, the functions that we write are specified to work on *infinite* domains (e.g., the set of all integers; the set of all lists of integers), but our test cases will only ever cover a finite number of possibilities.

That doesn't mean that we should give up on testing altogether! A good test suite can give us extremely high confidence that our code is correct.

So far the code we've written has been fairly straightforward, and we have focused our learning on understanding the basic building blocks of the Python programming language itself. But as you gain mastery of the language and progress in your study of computer science, you will encounter code that relies on abstract—often mathematical—properties of the data it is operating on to ensure its correctness.

A tale of two implementations

To illustrate this point, here's a simple example of two different implementations of a function that computes the sum of the first n positive integers:

```
def sum_to_n_v1(n: int):
    """Return the sum of the integers from 1 to n, inclusi

    Preconditions:
        - n >= 1

    >>> sum_to_n_v1(4)
    10
    >>> sum_to_n_v1(10)
    55
    """
    return sum([i for i in range(1, n + 1)])


def sum_to_n_v2(n: int):
    """Return the sum of the integers from 1 to n, inclusi

    Preconditions:
        - n >= 1

    >>> sum_to_n_v2(4)
    10
    >>> sum_to_n_v2(10)
    55
    """
    return n * (n + 1) // 2
```

Both of these functions have the same *function specification*, but they have two very different implementations. And we claim that both of these implementations are correct! To verify this, we could run the doctests, and create additional unit and property-based tests. But, you likely wouldn't find this very satisfying. The first implementation, in `sum_to_n_v1`, is "obviously" correct: its body is literally a translation of the mathematical expression $\sum_{i=1}^{n} i$ into Python code. On the other hand, the second implementation, in `sum_to_n_v2`, seems to have nothing to do with sums at all!

At this point you may want very much to interrupt us: *Come on, Mario and David, something something formula something something.* And indeed, if you're thinking this you're right: the reason the second implementation is correct is because of the following theorem we have from mathematics:

> **Theorem.** For all $n \in \mathbb{Z}^+$, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

So `sum_to_n_v2` computes and returns the value $\frac{n(n+1)}{2}$, and this theorem tells us that this value is equal to the sum of the first n positive integers. In other words, the correctness of `sum_to_n_v2` is justified not just by looking at the code or running test cases, but by this theorem. Now, some of you may have seen this theorem before, or been taught a "formula" for this summation expression. The point of this example is not to see whether you knew this bit of mathematics before reading this section—it is to illustrate the point that mathematical statements like this can influence the code that we write.

A preview: running time

After reading through the previous example, you might wonder why we bother having two different implementations at all: isn't enough to just stick with the more "obvious" definition in `sum_to_n_v1` and not have to worry about knowing this mathematical fact to understand `sum_to_n_v2`?

Indeed, a lot of the time we do choose the implementation that most closely resembles the written definitions/descriptions in the function specification, because it is the easiest to understand. However, there are good reasons to use alternate implementations that rely on more mathematics or other domain-specific knowledge: computation speed.

Try calling the two versions of this function on a very large number like 10^8:

```
>>> sum_to_n_v1(10 ** 8)
5000000050000000
>>> sum_to_n_v2(10 ** 8)
5000000050000000
```

Even though both versions return the same value, you should have noticed a difference—the first version takes noticeably longer to return than the second. Computers have gotten so fast that we're used to calculations happening instantaneously, but of course every calculation takes some amount of time, and the more calculations that a function performs, the longer it takes.

So our second version of this function does come with a significant benefit: by using this mathematical theorem, it is able to calculate a sum of a very large sequence of numbers almost instantaneously. In other words, this theorem gives us a way to speed up this computation, replacing a straightforward-but-slow implementation with a faster one. In future chapters, we'll study how to formally analyse the *running time* of a function. For now, you just need to know that while our two implementations have roughly the same length (measure in number of lines of code), one is much, much slower than the other.

Though we're only introducing these ideas in this chapter with relatively simple examples, the idea of *swapping one implementation for another* will come up again and again in this course. As the functions and programs you write grows larger, efficiency will be an important consideration for your code, and so it will be common to start with one function implementation and eventually replace it with another.