Integer Overflow

When you create an integer variable, your computer allocates a fixed number of bits for storing it. Most modern computers use 32 or 64 bits. But some numbers are *so big* they don't fit even in 64 bits, like sextillion (a *billion trillion*), which is 70 digits in binary.

Sometimes we might have a number that *does* fit in 32 or 64 bits, but if we add to it (or multiply it by something, or do another operation) the result might *not fit* in the original 32 or 64 bits. This is called an **integer overflow**.

For example, let's say we have just **2** bits to store integers with. So we can only hold the unsigned (non-negative) integers 0-3 in binary:

```
00 (0)
01 (1)
10 (2)
11 (3)
```

What happens if we have 3 (11) and we try to add 1 (01)? The answer is 4 (100) but that requires 3 bits and we only have 2.

What happens next depends on the language:

- Some languages, like Python or Ruby, will notice that the result won't fit and automatically allocate space for a larger number.
- In other languages, like Objective C or Java, the processor will sort of "do its best" with the bits it has, taking the true result and throwing out any bits that don't fit. So in our example above, when adding 01 to 11, the processor would take the true result 100 and throw out the highest bit, leaving 00.

- Swift will throw an error if an integer overflows, unless you've explicitly indicated that overflowing integers should be truncated to fit (like in Objective C or Java).

In languages where integer overflow can occur, you can reduce its likelihood by using larger integer types, like Objective C's `long long int` or Java's `long`. If you need to store something even bigger, there are libraries built to handle arbitrarily large numbers.

In some languages, you can also take advantage of overflow-checking features provided by the compiler or interpreter.

⬅ course home (/table-of-contents)

Next up: The Stolen Breakfast Drone ➡ (/question/find-unique-int-among-duplicates?course=fc1&section=bit-manipulation)