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 requres 3 bits and we only have 2!

In a high-level language like Python, the interpreter will notice this is about to happen and switch to a larger integer type automatically. But, in a lower-level language like C, 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. This can cause some pretty nasty bugs.

In some languages, you can reduce the risk of integer overflow by using larger integer types (like `long long int` in C and C++ or `BigInteger` in Java). Other modern languages will automatically use larger integer types if your program needs them (like `long` in Python and `Bignum` in Ruby).

What's next?

If you're ready to start applying these concepts to some problems, check out our mock coding interview questions (/next).

They mimic a real interview by offering hints when you're stuck or you're missing an optimization.

Try some questions now ➜

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.