Set 2. Limitations of Storing Numbers

Skill 2.01: Explain how computers store positive and negative integers

Skill 2.01 Concepts

When computer programs store numbers in variables, the computer needs to find a way to represent that number in computer memory. Computers use different strategies based on whether a number is an integer or not. Due to limitations in computer memory, programs sometimes encounter issues with **round-off**, **precision**, or **overflow** of numeric values.

Integer Representation

An **integer** is any number that can be written without a fractional component. The same term is used in both programming and in math, so hopefully it's familiar to you.

All of these numbers are integers: 120, 10, 0, -10

How can a programming language represent those integers in computer memory? Well, computers represent *all data* with bits, so we know that ultimately, each of those numbers is a sequence of 0s and 1s.

To start simple, let's imagine a computer that uses only 4 bits to represent integers. It can use the first bit to represent the sign of the integer, positive or negative, and the other 3 bits for the absolute value.

In that system, the number 1 would be represented like this:

0	0	0	1
+/-	4	2	1
sign	2^2	2^1	2^0

The 0 in the sign bit represents a positive number, and the 1 in the right most bit represents the 2^0 (1) place of the value.

What's the largest number this system could represent? Let's fill all the value bits with 1 and see,

0	1	1	1
+/-	4	2	1
sign	2^2	2^1	2^0

That’s the positive number 7, since 22 + 21 + 20 = (4 + 2 + 1) = 7

Skill 2.01 Exercise 1

Skill 2.02: Explain the cause of “overflow” errors

Skill 2.02 Concepts

What would happen if we ran a program like this on the 4-bit computer, where the largest positive integer is 7?

```
var x = 7;
var y = x + 1;
```

The computer can store the variable x just fine, but y is one greater than the largest integer it can represent with the 4 bits. In a case like this, the computer might report an "overflow error" or display a message like "number is too large". It might also truncate the number (capping all results to 7) or wrap the number around (so that 8 becomes 1).

We don't want to end up in any of those situations, so it's important we know the limitations of our language and environment when writing programs.

Fortunate, most modern computers use 64-bit architectures which can store incredibly large integers. In JavaScript, the largest safe integer is 9,007,199,254,740,922, equivalent to 252-1. Beyond that, and we are in the danger zone.

Consider the example below

https://studio.code.org/projects/applab/Fzcr8TXivJ4t7NsjETmAZLf-JG31Q1tcViPyjfD0lNQ/view

Skill 2.02 Exercise 1

Skill 2.03: Explain how computers store floating point numbers

Skill 2.03 Concepts

We've seen there are limitations to storing integers in a computer. Numbers that aren't integers, like fractions and irrational numbers, are even trickier to represent in computer memory.

Consider numbers like 2/5, 1.234, 9.99999999, or the famously never-ending π

Computer languages typically use floating-point representation for non-integers (and sometimes integers too). It’s similar to “scientific notation”, a representation you might know from other subjects.

In floating-point representation, a number is multiplied by a base that’s raised to an exponent,

$$300 = 3 \times \underbrace{10^{\overbrace{2}^{\text{exponent}}}}_{\text{base}}$$

Since computers use the binary system instead of the decimal system, the base for floating-point numbers is 2 instead of 10. Because of that, numbers that are exactly powers of 2 are the simplest to represent.

$$128 = 1 \times 2^7$$
$$256 = 1 \times 2^8$$

Numbers between powers of 2 look like this,

$$160 = 1.25 \times 2^7$$
$$192 = 1.50 \times 2^7$$
$$224 = 1.75 \times 2^7$$

What about non-integers? Once again, powers of 2 are the simplest to represent,

$$0.50 = 1 \times 2^{-1}$$
$$0.25 = 1 \times 2^{-2}$$

Floating-point can also represent fractions between powers of 2:

$$0.750 = 1.5 \times 2^{-1}$$
$$0.375 = 1.5 \times 2^{-2}$$

Once the computer determines the floating point representation for a number, it stores that in bits. Modern computers use a 64-bit system that uses 1 bit for the sign, 11 bits for the exponent, and 52 bits for the number in front.

Here is 0.375 in that binary floating-point representation,

0011111111011000000000000000000000000000000000000000000000000000

The exact translation to bits is more complicated than we can go into here, but is a great topic for those of you who want to dive deeper.

Skill 2.03 Exercise 1

Skill 2.04: Explain the cause of round-off errors

Skill 2.04 Concepts

Floating-point representation still can't fully represent all numbers, however, consider the fraction 1/3 and its floating point representation,

$$1/3 = 1.\overline{3} \times 2^{-2}$$

In binary, .3 is an infinitely repeating sequence,

0101010101…

We cannot store an infinite sequence in a computer! At some point, the computer has to end the number somehow either by chopping it off or rounding to the nearest floating point number. Computers have to do that fairly often, as even fractions like 1/20 (which is a short 0.1 in decimal) end up as infinitely repeating sequences once converted to binary.

We often don't notice the lower precision of a number's representation until we use it in calculations. That's when we experience a round-off error in the results.

Below is a program that attempts to add 0.1 + 0.1 + 0.1. In the non-computer world, we know that is 0.3, but in the computer, each of the 0.1 values is stored as a rounded-off binary fraction, and when they are added together, they do not quite equal what we expect…

https://studio.code.org/projects/applab/aTo8WzMW8HdK11UB0GVYobJWgD3kCawG_J_q9MZ_ePI/view

The more bits we can use, the more precise our numbers and calculations will be. Modern 64-bit systems offer a high enough precision for low-stakes calculations.

Perhaps at some point in your life, you will find yourself writing programs that calculate voting outcomes, power a self-driving car, or even launch a rocket. When the stakes are high, precision matters!

Skill 2.04 Exercises 1 & 2