Review. Possibilities for binary representations of real numbers:

- fixed-point number: $\pm x.y$ with x and y of a certain number of bits
- floating-point number: $\pm 1.x \cdot 2^y$ with x and y of a certain number of bits

  IEEE 754, single precision: 32 bit (1 bit for sign, 23 bit for significand x, 8 bit for exponent y)

  IEEE 754, double precision: 64 bit (1 bit for sign, 52 bit for significand x, 11 bit for exponent y)

Example 12. Represent -0.375 as a single precision floating-point number according to IEEE 754.

Solution. $-0.375 = -\frac{3}{8} = -3 \cdot 2^{-3} = \boxed{-}\underbrace{1.\boxed{1}}_{\text{binary}} \cdot 2^{-2}$

The exponent -2 gets stored as $-2+127 = \boxed{0111,1101}$. (Recall that the bias $2^7-1=127$ is being added to the exponents. Also, it helps to keep in mind that $127=(0111,1111)_2$.)

Overall, -0.375 is stored as $\boxed{1}\boxed{0111,1101}\boxed{1000,0000,\ldots}$.

Example 13. What is the largest single precision floating-point number according to IEEE 754?

Technical detail. The exponent $(1111,1111)_2 = 2^8-1=255$ is reserved for special numbers (such as infinities or "NaN"). Hence, the largest exponent corresponds to $(1111,1110)_2=254$.

Solution. The largest single precision floating-point number is

$$\boxed{1}\boxed{1111,1110}\boxed{1111,1111,\ldots} = +1.111\ldots \cdot 2^{127} \approx 2^{128} = 2^{2^7} \approx 3.4 \cdot 10^{38}.$$

Here, we used that $(1111,1110)_2 = 2^8-2=254$ so that the actual exponent is $254-127=127$.

For comparison. The largest 32 bit (signed) integer is $2^{31}-1 \approx 2.1 \cdot 10^9$ (the exponent is $31=32-1$ to account for using 1 bit to store the sign). You might find this surprisingly small. And, indeed, 32 bit is not enough to address all memory locations in modern systems which is why the step to 64 bit was necessary.

Double precision. Likewise, the largest double precision floating-point number is

$$\boxed{1}\boxed{111,1111,1110}\boxed{1111,1111,\ldots} = +1.111\ldots \cdot 2^{1023} \approx 2^{1024} = 2^{2^{10}} \approx 1.8 \cdot 10^{308}.$$

On the other hand, the largest 64 bit (signed) integer is $2^{63}-1 \approx 9.2 \cdot 10^{18}$.

Example 14. (reasons for floats) Almost universally, major programming languages use floating-point numbers for representing real numbers. Why not fixed-point numbers?

Solution. Fixed-point numbers have some serious issues for scientific computation. Most notably:

- Scaling a number typically results in a loss of precision.

  For instance, dividing a number by 2^r and then multiplying it with 2^r loses r digits of precision (in particular, this means that it is computationally dangerous to change units). Make sure that you see that this does not happen for floating-point numbers.

- The range of numbers is limited.

  For instance, the largest number is on the order of 2^N where N is the number of bits used for the integer part. On the other hand, a floating-point number can be of the order of $2^{2^{M-1}}$ where M is the number of bits used for the exponent. (Make sure you see how enormous of a difference this is! See the previous example for the case of double precision.)

Moreover, as noted in the box below, fixed-point numbers do not really offer anything that isn't already provided by integers. This is the reason why most programming languages don't even offer built-in fixed-point numbers.

> Fixed-point numbers are essentially like integers.
>
> For instance, instead of 21.013 (say, seconds) we just work with 21013 (which now is in milliseconds).

Armin Straub
straub@southalabama.edu

Example 15. Give an example where one should not use floats.

Solution. Most notably, one should not use floats when dealing with money. That is because, as we saw earlier, an amount such as 0.10 dollars cannot be represented exactly using a float (when using base 2, as is the default in most programming languages such as Python) and thus will get rounded. This is very problematic when working with money.

Comment. For most purposes, the easiest way to avoid these issues is to store dollar amounts as cents. For the latter we can then simply use integers and work with exact numbers (no rounding).

Example 16. `Python` Python automatically uses (double precision) floats when we enter numbers with a decimal point or as the result of divisions.

```
>>> 1.1
```

```
1.1
```

```
>>> 1/3
```

```
0.3333333333333333
```

Comment. Note how we can see (roughly) the 52 bit precision of the double precision floats (there are 16 decimal digits after the decimal point, which translates to about $16 \cdot \log_2 10 \approx 53.15$ binary digits; this corresponds to about 52 bit after the first implicit 1).

IMPORTANT. As noted earlier, the commands here are entered into an interactive Python interpreter (this is indicated by the >>>). When running a Python script, we need to use `print(1.1)` or `print(1/3)` to receive the above outputs.

For very large (or very small) numbers, scientific notation is often used:

```
>>> 2.0 * 10**80
```

```
2e+80
```

```
>>> (1/2)**100
```

```
7.888609052210118e-31
```

The following are two things that are (somewhat) special to Python and are often handled differently in other programming languages. First, integers are not limited in size (often, integers are limited to 64 bits, which can cause issues like overflow when one exceeds the 2^{64} possibilities). This is illustrated by the following (this explains why we wrote 2.0 above):

```
>>> 2 * 10**80
```

```
200000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Second, Python likes to throw errors when a computation runs into an issue (there are nice ways to "catch" these errors in a program and to react accordingly, but that is probably beyond what we will use Python for).

```
>>> 1 / 0
```

```
ZeroDivisionError: division by zero
```

Some other programming languages would instead (silently, without error messages) return special floats representing $+\infty$, $-\infty$ or `NaN` (not-a-number).

Example 17. To store 13 decimal digits, how many bits are needed?

Solution. There are 10^{13} many possibilities with 13 decimal digits. Since $\log_2(10^{13}) = 13\log_2(10) \approx 43.19$, we need 44 bits.

Armin Straub
straub@southalabama.edu

Example 18. Python Let us perform the previous calculation of $13\log_2(10)$ using Python. First of all, we need to get access to the $\log$ function because it is not available by default. Instead it resides in a module called `math`:

```
>>> from math import log
```

```
>>> 13*log(10, 2)
```

```
43.18506523353572
```

It might be unexpected that the 2 is the second argument of $\log$. If you want to learn more about how to use a function, you can enter the function name followed by a question mark:

```
>>> log?
```

Example 19. Python Explain the following floating-point rounding issue:

```
>>> 0.1 + 0.1 + 0.1 == 0.3
```

```
False
```

```
>>> 0.1 + 0.1 + 0.1
```

```
0.30000000000000004
```

Solution. As we saw in Example 6, 0.1 cannot be stored exactly as a floating-point number (when using base 2). Instead, it gets rounded up slightly. After adding three copies of this number, the error has increased to the point that it becomes visible as in the above output.

IMPORTANT. In the Python code above, we used the operator == (two equal signs) to compare two quantities. Note that we cannot use = (single equal sign) because that operator is used for assignment (`x = y` assigns the value of `y` to `x`, whereas `x == y` checks whether `x` and `y` are equal).

Comment. As the above issue shows, we should never test two floats x and y for equality. Instead, one typically tests whether the difference $|x-y|$ is less than a certain appropriate threshold. An alternative practical way is to round the floats before comparison (below, we round to 8 decimal digits):

```
>>> round(0.1 + 0.1 + 0.1, 8) == round(0.3, 8)
```

```
True
```