Exercise 1: IEEE 754 Limits

Consider a system of IEEE 754-like floating-point machine numbers with 10 bits, where the first bit is the sign bit, the next $E = 4$ bits are the exponent, and the last $M = 5$ bits are the mantissa. As is typical for floating-point representations, the bias of the exponent is $2^{E-1} - 1$, and special values are represented equivalently to the IEEE 754 standard.

(a) Determine the value of the smallest positive number (both normalized and denormalized), the biggest positive denormalized number, and the biggest finite number represented in this system. What are each of their binary representations?

(b) What are the binary representations of the numbers 0, ∞, and `NaN`? How many `NaN`s are there?

(c) Write down the binary representations of the number 1, the smallest representable number larger than 1, and the largest representable number smaller than 1. What is the value of each of these numbers? What do you observe about the binary representations?

(d) Determine the number of normalized numbers in this system.

(e) Determine the number of denormalized numbers in this system.

Exercise 2: Decimal to Binary Conversion

Converting a decimal number $xxx.yyyy_{10}$ to a binary number involves two parts:

Integer Part Repeatedly divide the integer part xxx by 2. The remainders, read in reverse order, form the binary integer representation.

Fractional Part Repeatedly multiply the fractional part $yyyy$ by 2. The integer part of the result (0 or 1) is the next binary digit after the point. Remove the integer part and continue with the remaining fractional part. Repeat until the fractional part becomes 0, a pattern repeats, or sufficient precision is achieved.

(a) Represent the number **18.6** in the binary system with at least seven bits after the binary point (so you get a number $\texttt{...xxxxx.yyyyyyy}_2$ with at least seven `y`). What do you notice?

(b) Consider a system of IEEE 754-like floating-point machine numbers with 14 bits, with the first bit for the sign, the next $E = 5$ bits for the exponent, and the last $M = 8$ bits for the mantissa. As is typical for floating-point representations, the exponent is biased by $2^{E-1} - 1$.

(i) What is the "machine epsilon" of this system?

(ii) Represent the number **18.6** as a binary machine number with $M = 8$ using your result from (a) by

- Truncation
- Rounding to the nearest.

What are the corresponding values (in decimal)?

(iii) What are the absolute and relative errors in subtask (b)ii? How do they relate to the machine epsilon?

Exercise 3: Floating Point Numbers in R

The builtin R function `numToBits` converts a numeric vector to a "`raw`" vector containing the binary representation of the numbers, with bits ordered from least to most significant. For example, $1.125 = 1 + 2^{-3}$ is represented as

```
numToBits(1.125)

#>  [1] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
#> [26] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
#> [51] 00 00 01 01 01 01 01 01 01 01 01 01 00 00
```

You can see the sign bit (at the very end) is `0`, the exponent 1 is represented as `01111111111` (in reverse order), and the significand 1.001000... is represented as `001000...`. Note that `raw` vectors can be converted to `logical`s, `integer`s, and `numeric`s by using the typical R type conversion functions `as.logical()` etc.

Write a function `r()` (short for "representation", single letter for convenience) that takes a numeric scalar and prints it in the format `S 2^EEE * F.FFFFFF...`, where `S` is the sign ("`+`" or "`-`"), `EEE` is the exponent (in decimal) of the scaling factor, and `F.FFFFFF...` is the significand (in binary, including the implicit/hidden bit, and including all trailing zeros to 52 digits after the binary point). Your function should handle normalized and denormalized numbers, as well as ±0. The scaling factor of denormalized numbers should be displayed as `2^-1022`.

```
r(1.125)  # typical case
```

```
#> + 2^0 * 1.0010000000000000000000000000000000000000000000000000
```

```
r(-1.125)  # negative number
```

```
#> - 2^0 * 1.0010000000000000000000000000000000000000000000000000
```

```
r(22)
```

```
#> + 2^4 * 1.0110000000000000000000000000000000000000000000000000
```

```
r(0)  # positive zero
```

```
#> + 0 * 0.0000000000000000000000000000000000000000000000000000
```

```
r(-0)  # negative zero
```

```
#> - 0 * 0.0000000000000000000000000000000000000000000000000000
```

```
r(8e-323)  # denormalized number
```

```
#> + 2^-1022 * 0.0000000000000000000000000000000000000000000000010000
```

When trying to understand floating point numbers, this function is very helpful. We encourage you to play around with it, e.g. in the next exercise sheet.