Exercise 1: Two's Complement Basics

Solve the following tasks using two's complement representation in an 8-bit computer:

(a) Represent the numbers 55, 8 and -8 in $\mathbb{Z}$ as machine numbers.

(b) Calculate $55 - 8$ in the 8-bit binary system and convert the result into a decimal number to check your result. Can you come up with two distinct ways of doing this?

Exercise 2: Binary to Numeric Conversion

Write an R function `binaryToNumber(binary)` that takes a binary machine number (in the form of `numeric` vector of "1"s and "0"s) in two's complement representation and outputs the corresponding integer as a single `numeric` value. Consider the number to be negative if the first bit is "1":

```
binaryToNumber(c(0, 0, 0, 1, 0, 1, 1))
```

```
#> [1] 11
```

```
binaryToNumber(c(1, 1, 0, 1))
```

```
#> [1] -3
```

Exercise 3: Binary Division

Binary division, i.e. division of two binary integers, works very similar to the long division method used for decimal numbers that you are familiar with from school. Here we are only considering positive integers.

Convert the decimal numbers 120 and 13 into their 8-bit binary representations. Perform binary long division to calculate $120 \div 13$. Show your steps. What is the quotient (i.e., the result of the division) and the remainder? *Hint:* Binary long division works just like decimal long division: you compare the current segment of the dividend with the divisor, subtract if possible (writing a '1' in the quotient), or don't subtract (writing a '0' in the quotient), and then bring down the next bit from the dividend.

Exercise 4: Character Encoding

Computers fundamentally store information as sequences of bits, often grouped into bytes (8 bits). To represent text, we need a way to map characters (e.g. letters, digits, symbols, emojis) to numeric values and to then encode these values into bytes.

Unicode is a standard that assigns a unique number, called a **code point**, to virtually every character used in modern writing systems worldwide. Code points are typically represented in the format `U+` followed by hexadecimal digits (e.g., `U+0041` for code point 65, i.e. 'A', `U+20AC` for code point 8364, i.e. '€' etc.). The highest code point is `U+10FFFF`, which is $2^{20} + 2^{16} - 1 = 1114111$.[1]

While Unicode defines the code points, it doesn't specify how these (potentially large) numbers are stored as byte sequences. This is the job of **character encoding schemes**. A very common scheme is **UTF-8**, which is a variable-length encoding (see also the lecture notes for more details):

- Code points in the ASCII range (U+0000 to U+007F) are encoded using a single byte.
- Higher code points are encoded using sequences of 2, 3, or 4 bytes.

This makes UTF-8 backward-compatible with ASCII and efficient for texts that primarily use ASCII characters, while still being able to represent all Unicode code points.

[1] This may appear somewhat arbitrary and is a historical artifact.

(a) **Implement UTF-8 Conversion Functions:**
Write two R functions:

- `codepointsToUtf8(codepoints)`: Takes an integer vector of Unicode code points and returns an integer vector representing the corresponding UTF-8 byte sequence (values between 0 and 255). Note that this should neither handle, nor emit, `character` or `raw` vectors, but only `integer` or integer-valued `numeric` vectors.
- `utf8ToCodepoints(bytes)`: Takes an integer vector of UTF-8 bytes (values 0-255) and returns an integer vector of the decoded Unicode code points. Your function should handle multi-byte sequences correctly. You don't need to implement robust error handling for invalid byte sequences for this exercise.

Do not use packages or R's own conversion functions (e.g. `iconv`, `utf8ToInt`, `intToUtf8`, etc.) for this exercise. You may use some bit operations provided by base R, e.g. `bitwShiftL`, `bitwShiftR`, `bitwAnd`, `bitwOr` (generally: type in `bitw` and press `Tab` to see these options). Test your own functions by making sure that `utf8ToCodepoints(codepointsToUtf8(x))` returns `x` for any vector of integers between 0 and `U+10FFFF` $= 2^{20} + 2^{16} - 1$.

(b) **Explore Character Representation:**

(i) Create a plain text file (e.g., `chars.txt`).

(ii) Copy and paste the following characters into your file, each separatedby a space. If copy/pasting from this PDF does not work, go to a resource like `https://emojipedia.org` or use your system's character map/emoji picker.

- The capital letter 'X'
- The German umlaut 'ä'
- The "Fire" emoji (🔥)
- The "Thumbs Up: Light Skin Tone" emoji (👍🏻)
- The "Family: Woman, Girl" emoji (👩‍👧)

(Only copy the characters, not the text comments. Your file should read "`X ä` 🔥 👍🏻 👩‍👧".)

(iii) Save the file using UTF-8 encoding (this is usually the default in modern text editors) as `chars.txt`.

(iv) In R, read the raw bytes from this file using:
`bytes <- readBin("chars.txt", "integer", n = 100, size = 1, signed = FALSE)`
(Ensure the file path is correct. The `n` sets an upper limit on the number of bytes read).

(v) Use your `utf8ToCodepoints` function from part (a) to convert the byte vector back into Unicode code points. Print the hexadecimal code points using `sprintf("U+%04X", codepoints)`.

(vi) Look up the code points reported by your function (e.g., using `https://unicodeplus.com/` or similar resources). What do you observe about the relationship between the perceived characters/emojis and the number of code points used to represent them? Pay attention to the emojis with modifications (skin tone) or combinations (family).