Solution 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?

Representation of a machine number with 32 bits: $x = \sum_{i=1}^{31} u_i 2^{i-1} - u_{32} 2^{31}$ $\Rightarrow$ With 8 bits: $x = \sum_{i=1}^{7} u_i 2^{i-1} - u_8 2^7$

(a)

- Decomposition of the number $55 = 32 + 16 + 4 + 2 + 1 = 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$:

	Bit							
	u_8	u_7	u_6	u_5	u_4	u_3	u_2	u_1
55_{10}	0	0	1	1	0	1	1	1
	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Alternatively: 7 bits for representing the number, 1 bit for the sign (R = "remainder")

2^0	55	/	2	=	27	R	1
2^1	27	/	2	=	13	R	1
2^2	13	/	2	=	6	R	1
2^3	6	/	2	=	3	R	0
2^4	3	/	2	=	1	R	1
2^5	1	/	2	=	0	R	1
2^6	0	/	2	=	0	R	0

$\Rightarrow$ Sign: 0, 7-bit number: 0110111 $\Rightarrow$ Total = 00110111.

- Number 8:

2^0	8	/	2	=	4	R	0
2^1	4	/	2	=	2	R	0
2^2	2	/	2	=	1	R	0
2^3	1	/	2	=	0	R	1
2^4	1	/	2	=	0	R	0
$\vdots$							$\vdots$

$\Rightarrow$ Sign: 0, 7-bit number: 0001000. Total = 00001000.

- Number -8 using two's complement, i.e.:

		Bit							
		u_8	u_7	u_6	u_5	u_4	u_3	u_2	u_1
$\lvert -8_{10} \rvert$		0	0	0	0	1	0	0	0
invert		1	1	1	1	0	1	1	1
	+								1
add 1		1	1	1	1	1	0	0	0
		-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$$\Rightarrow -2^7 + 2^6 + 2^5 + 2^4 + 2^3 = -8$$

The leading bit is thus added with a negative value.
Advantage: You save one bit, since there are not two representations for "0".

(b)
- Option 1: Represent subtraction as addition: $55 - 8 = 55 + (-8)$

$$\begin{array}{rcccccccc} & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ + & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \end{array}$$

$\Rightarrow$ Overflow, leading 1 is dropped since only 8 bits are available.
$\Rightarrow$ Converting the result 00101111_2 to the decimal system: $2^5+2^3+2^2+2^1+2^0 = 32+8+4+2+1 = 47$
The result matches $55 - 8 = 47$.

- Option 2: Subtract (possibly with borrowing): We calculate $55 - 8$:

$$\begin{array}{rc} \text{Borrow} & {\scriptstyle 1} \\ & 00110111 \\ - & 00001000 \\ \hline \text{Result} & 00101111 \end{array}$$

Explanation: We perform binary subtraction column by column from right to left. $1-0=1$, $1-0=1$, $1-0=1$. For the next column (2^3), we calculate $0-1$. We need to borrow from the left (2^4). This makes the current column $(0+2)-1=1$. The column we borrowed from (2^4) becomes $1-1=0$. The remaining columns are $1-0=1$, $0-0=0$, $0-0=0$.
The result is 00101111_2 as above.

Solution 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
```

Encoding of integers in two's complement as follows:

$$\Rightarrow (\sum_{i=1}^{n-1} u_i \cdot 2^{i-1}) - u_n \cdot 2^{n-1}$$

Our function should therefore return 7 for e.g. "0111" (index starts at the right!).

Using an explicit loop:

```
binaryToNumber <- function(binary) {
  # machine number to vector
  checkmate::assertIntegerish(binary, lower = 0, upper = 1, tol = 0, any.missing = FALSE)
  if (!length(binary)) return(0L)

  # calculate integer over loop
  result <- 0L
  binary.rev <- rev(binary)
  #apply formula
```

```
  for (i in seq_along(binary)) {
    result = result + binary.rev[i] * 2L^(i-1L)
  }
  # Correct two's complement adjustment: subtract 2 * sign_bit * 2^(n-1)
  n <- length(binary)
  sign.bit <- binary.rev[[n]] # Or binary[[1]]
  unsigned.val <- result
  unsigned.val - 2L * sign.bit * 2L^(n-1L)
}
```

Using a vectorized approach:

```
binaryToNumber <- function(binary) {
  # machine number to vector
  checkmate::assertIntegerish(binary, lower = 0, upper = 1, tol = 0, any.missing = FALSE)
  if (!length(binary)) return(0L)

  n <- length(binary)
  powers.of.2 <- 2L^((n-1L):0) # Powers from 2^(n-1) down to 2^0
  unsigned.val <- sum(binary * powers.of.2)

  # Correct two's complement adjustment
  sign.bit <- binary[[1]]
  unsigned.val - 2L * sign.bit * 2L^(n-1L)
}
```

Example calls:

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

#> [1] 11

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

#> [1] -3

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

#> [1] 11

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

#> [1] -3
```

Solution 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.

First, convert the decimal numbers 120 and 13 to binary.

- $120_{10} = 64 + 32 + 16 + 8 = 2^6 + 2^5 + 2^4 + 2^3 = 1111000_2$.
  As an 8-bit number: 01111000_2.

- $13_{10} = 8 + 4 + 1 = 2^3 + 2^2 + 2^0 = 1101_2$.
  As an 8-bit number: 00001101_2.

Now, perform binary long division for $1111000 \div 1101$:

```
        1001  <- Quotient
      -------
1101 | 1111000 <- Dividend
     - 1101    (Compare 1111 >= 1101? Yes. Q=1. Subtract 1101)
     -------
        0010   (Result 10)
         0   (Bring down 0 -> 100. Compare 100 < 1101? Yes. Q=0)
          0  (Bring down 0 -> 1000. Compare 1000 < 1101? Yes. Q=0)
           0 (Bring down 0 -> 10000. Compare 10000 >= 1101? Yes. Q=1. Subtract 1101)
         - 1101
         -------
           0011 <- Remainder
```

The quotient is 1001_2, which is $2^3 + 2^0 = 8 + 1 = 9$ in decimal. The remainder is 0011_2, which is $2^1 + 2^0 = 2 + 1 = 3$ in decimal. Check: $13 \times 9 + 3 = 117 + 3 = 120$.

Solution 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.

(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:**

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

(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).

(a) Here are R implementations for converting between Unicode code points and UTF-8 byte sequences. We use bitwise operations provided by base R (`bitwShiftL`, `bitwShiftR`, `bitwAnd`, `bitwOr`) to manipulate the bits according to the UTF-8 encoding rules.

```r
#' Converts a vector of Unicode code points to a vector of UTF-8 bytes.
#' @param codepoints An integer vector of Unicode code points.
#' @return An integer vector of UTF-8 bytes (0-255).
codepointsToUtf8 <- function(codepoints) {
  checkmate::assertIntegerish(codepoints, lower = 0, any.missing = FALSE)

  makeContinuationByte <- function(value, shift) {
    bitwOr(0x80, bitwAnd(bitwShiftR(value, shift), 0x3F))
  }

  bytes.list <- lapply(codepoints, function(cp) {
    if (cp < 0x80) { # 1-byte sequence (ASCII)
      # 0xxxxxxx -> 00000000 00000000 00000000 00000xxx
      cp
    } else if (cp < 0x800) { # 2-byte sequence
      # 00000xxx xxyyyyyy -> 110xxxxx 10yyyyyy
      c(
        bitwOr(0xC0, bitwShiftR(cp, 6)),                  # 110xxxxx
        makeContinuationByte(cp, 0)                       # 10yyyyyy
      )
    } else if (cp < 0x10000) { # 3-byte sequence
      # xxxxyyyy yyzzzzzz -> 1110xxxx 10yyyyyy 10zzzzzz
      c(
        bitwOr(0xE0, bitwShiftR(cp, 12)),                 # 1110xxxx
        makeContinuationByte(cp, 6),                      # 10yyyyyy
        makeContinuationByte(cp, 0)                       # 10zzzzzz
      )
    } else if (cp <= 0x10FFFF) { # 4-byte sequence
      # 000uuuxx xxxxyyyy yyzzzzzz -> 11110uuu 10xxxxxx 10yyyyyy 10zzzzzz
```

```
      c(
        bitwOr(0xF0, bitwShiftR(cp, 18)),                 # 11110www
        makeContinuationByte(cp, 12),                     # 10xxxxxx
        makeContinuationByte(cp, 6),                      # 10yyyyyy
        makeContinuationByte(cp, 0)                       # 10zzzzzz
      )
    } else {
      stop("Invalid Unicode code point: ", sprintf("U+%X", cp))
    }
  })
  # Combine list of byte vectors into a single vector
  unlist(bytes.list)
}

# Example usage:
codepointsToUtf8(c(0x41, 0xE4, 0x20AC, 0x1F525)) # A, ä, €, <Fire Emoji>
```

```
#>  [1]  65 195 164 226 130 172 240 159 148 165
```

```
#' Converts a vector of UTF-8 bytes to a vector of Unicode code points.
#' For simplicity, we assume that the input is valid UTF-8 and do not check validity.
#' @param bytes An integer vector of UTF-8 bytes (0-255).
#' @return An integer vector of Unicode code points.
utf8ToCodepoints <- function(bytes) {
  checkmate::assertIntegerish(bytes, lower = 0, upper = 255, any.missing = FALSE)

  codepoints <- integer(0)
  i <- 1L
  n <- length(bytes)

  while (i <= n) {
    b1 <- bytes[[i]]
    num.bytes <- 0L
    cp <- NA_integer_

    if (bitwAnd(b1, 0x80) == 0) { # 1-byte sequence (0xxxxxxx)
      # 0xxxxxxx stays as is
      cp <- bytes[[i]]
      i <- i + 1L
    } else if (bitwAnd(b1, 0xE0) == 0xC0) { # 2-byte sequence (110xxxxx)
      # 110xxxxx 10yyyyyy -> 00000xxx xxyyyyyy
      cp <-
        bitwShiftL(bitwAnd(bytes[[i]], 0x1F), 6) +
        bitwAnd(bytes[[i + 1]], 0x3F)
      i <- i + 2L
    } else if (bitwAnd(b1, 0xF0) == 0xE0) { # 3-byte sequence (1110xxxx)
      # 1110xxxx 10yyyyyy 10zzzzzz -> xxxxyyyy yyzzzzzz
      cp <-
        bitwShiftL(bitwAnd(bytes[[i]], 0x0F), 12) +
        bitwShiftL(bitwAnd(bytes[[i + 1]], 0x3F), 6) +
        bitwAnd(bytes[[i + 2]], 0x3F)
      i <- i + 3L
    } else if (bitwAnd(b1, 0xF8) == 0xF0) { # 4-byte sequence (11110xxx)
      # 11110www 10xxxxxx 10yyyyyy 10zzzzzz -> 000wwwxx xxxxyyyy yyzzzzzz
      cp <-
        bitwShiftL(bitwAnd(bytes[[i]], 0x07), 18) +
        bitwShiftL(bitwAnd(bytes[[i + 1]], 0x3F), 12) +
        bitwShiftL(bitwAnd(bytes[[i + 2]], 0x3F), 6) +
        bitwAnd(bytes[[i + 3]], 0x3F)
```

```
      i <- i + 4L
    } else {
      stop("Invalid UTF-8 sequence: ", paste(bytes[i:min(i + 3, n)], collapse = " "))
    }
    codepoints[[length(codepoints) + 1]] <- cp
  }
  codepoints
}

# Example usage (corresponding to previous example):
utf8.bytes <- c(0x41, 0xC3, 0xA4, 0xE2, 0x82, 0xAC, 0xF0, 0x9F, 0x94, 0xA5)
result.cps <- utf8ToCodepoints(utf8.bytes)
sprintf("U+%04X", result.cps) # Show as Hex

#> [1] "U+0041"  "U+00E4"  "U+20AC"  "U+1F525"
```

(b) Let's assume we created the file `chars.txt` with the content "X ä 🔥👍🤷" (or similar, perhaps separated by newlines) and saved it as UTF-8.

First, we read the bytes from the file. (Note: We need to create this file manually before running the code). The exercise tells us to just use `file.size` of 100, but the following would be the "proper" way to do it:

```
file.path <- "chars.txt"
file.size <- file.info(file.path)$size
char.bytes <- readBin(file.path, "integer", n = file.size, size = 1, signed = FALSE)
```

Written out manually, the bytes are:

```
  # UTF-8 bytes for: X ä ... (separated by space U+20)
  char.bytes <- c(0x58, 0x20,
    0xC3, 0xA4, 0x20,
    0xF0, 0x9F, 0x94, 0xA5, 0x20,
    0xF0, 0x9F, 0x91, 0x8D, 0xF0, 0x9F, 0x8F, 0xBB, 0x20,
    0xF0, 0x9F, 0x91, 0xA9, 0xE2, 0x80, 0x8D, 0xF0, 0x9F, 0x91, 0xA7)
```

Now, we use our function to convert these bytes back to code points:

```
# Decode the bytes using our function
decoded.codepoints <- utf8ToCodepoints(char.bytes)

# Display the results both as a vector of integers and in Hexadecimal format
print(decoded.codepoints)

#>  [1]     88     32    228     32 128293     32 128077 127995     32 128105
#> [11]   8205 128103

sprintf("U+%04X", decoded.codepoints)

#>  [1] "U+0058"  "U+0020"  "U+00E4"  "U+0020"  "U+1F525" "U+0020"  "U+1F44D"
#>  [8] "U+1F3FB" "U+0020"  "U+1F469" "U+200D"  "U+1F467"
```

Let's compare the output to the known Unicode code points, ignoring the space separator (U+20):

- **'X'**: Expected: `88` / `U+58`. This is a single ASCII character, represented by one byte and one code point.
- **'ä'**: Expected: `228` / `U+E4`. This is a single Latin character outside the ASCII range, represented by two bytes (`C3 A4`) in UTF-8 but still a single code point.
- **'🔥'**: Expected: `128293` / `U+1F525`. This is a single emoji, represented by four bytes (`F0 9F 94 A5`) in UTF-8 and corresponds to a single code point.
- **'👍🏻'**: Expected: `128077` / `U+1F44D` followed by `127995` / `U+1F3FB`. This single perceived emoji glyph is actually a sequence of **two** code points: the base emoji (Thumbs Up, `U+1F44D`) and the skin tone modifier (Light Skin Tone, `U+1F3FB`). UTF-8 uses 4 bytes for each (`F0 9F 91 8D` and `F0 9F 8F BB`).

- '👩‍👧': Expected: `128105 / U+1F469, 8205 / U+200D, 128103 / U+1F467`. This family emoji glyph is composed of **three** code points: Woman (`U+1F469`), a special *Zero Width Joiner* (ZWJ, `U+200D`), and Girl (`U+1F467`). The ZWJ tells rendering systems to combine the adjacent characters into a single glyph if possible. UTF-8 uses 4 bytes for Woman (`F0 9F 91 A9`), 3 bytes for ZWJ (`E2 80 8D`), and 4 bytes for Girl (`F0 9F 91 A7`).

Key Observation: What appears as a single character or symbol (a "glyph") on the screen does not always correspond to a single Unicode code point. Simple characters often do, but complex emojis, especially those involving modifications (like skin tone) or combinations (like families or flags), are represented by sequences of multiple code points. UTF-8 then encodes each of these code points into its corresponding byte sequence (1-4 bytes each). This highlights the difference between the abstract character (code point), its byte representation (encoding), and its visual appearance (glyph).