Practice 1: Text and Binary Files

The objective of this practice is to understand the differences between **text files** and **binary files**, focusing on how information is encoded, how it is processed using buffers, and how performance is affected by the chosen format and buffer size.

In this practice, the data consists of **integers**. Your program must read the data from a file, compute the **sum of all integers**, and write the result to **standard output**.
For the text case, the input file uses a **comma-separated values (CSV)** format.

Program usage

./myprogram binary|text pathToFile sizeOfTheBuffer

Where:

- `binary|text` specifies the format of the input file.
- `pathToFile` is the path to the input file.
- `sizeOfTheBuffer` is the buffer size expressed **in bytes**.

Binary file processing

In the binary case, the file contains integers stored in raw binary format. Since each element has a fixed size, reading is more efficient when the buffer size is a multiple of the integer size.
If necessary, you should adjust the buffer size so that only complete elements are read, even if this means using slightly fewer bytes per read.

Text file processing

In the text case, each integer has a **variable length**, and a read operation may split a number across multiple reads. Therefore, it is not guaranteed that each read contains complete values.

To correctly process the data, you must use a **circular buffer** to store incomplete data between reads. A circular buffer is a data structure that allows efficient insertion and removal of elements without reallocating memory. You can find more information here:

https://en.wikipedia.org/wiki/Circular_buffer

You are provided with an implementation of a circular buffer that can be used to store the data read from the file and to extract complete numbers from it. The buffer allows you to accumulate partial data across multiple read operations and process elements only when they are fully available.

The implementation includes functions to:

- Push/pop characters into/from the buffer.

- Determine the size of the next complete element stored in the buffer.

The function that determines the size of the next element returns:

- `-1` if no complete element is currently available in the buffer.
- The number of bytes of the next element if a complete element is present.

[Hint:] Before pushing or popping data, always ensure that the buffer is not full and not empty, respectively, to avoid data loss or invalid operations.

[Hint 2:] While it is possible to directly read from the file to the circular buffer, the logic is more complicated. The easiest version is to use a secondary linear buffer, with the same size, to where do read operations.

Tasks

1. Implement the program described above.
   Measure and compare the **wall-clock time** required to read and process the data for both binary and text files using the following buffer sizes: 16, 64, 128 and 1024 bytes. Create a table showing the results. To measure execution time, you may refer to *03_readFromTextFile_buffer.c, from the second theory class* or use the `time` command in the terminal (you can access the documentation and usage by typing `man time` in the terminal).

2. Answer the following questions:

   - Compare the first 10 numbers in the text file and the binary file. Are they the same? You may use `hexdump` to inspect the binary file, or write your own program to do so.
   - What is the size of each file?
   - What happens if you read a binary file as text?
   - What happens if you read a text file as binary?

Submission instructions

Submit all files through gradescope. Remember to add the team members during the submission process, and also a small text file including the team members names and NIA. Both team members should be from the same practice subgroup. The submitted files must be:

- All source (`.c`) and header (`.h`) files used, within a folder called `src`
- A `pdf` file answering the questions above. It should be no longer than a single page.
- A `txt` file with the names and NIA of the team members.
- A bash file with the terminal commands used to compile the sources.