Write your answers to the special response sheet you received (with your name and photograph). If you are using more than a single sheet of paper for your answers, then mark each sheet with its number / total number of sheets you will hand over.

Common part for tasks marked X

In **attachments 1** and **2** you will find the specification of the bitmap image format TARGA (files with the extension `.TGA`) - the format specification is available in the attachments in two complementary variants: 1) from Wikipedia, which is shorter and clearer, and 2) more detailed specification for finding details missing on Wikipedia.
Further in **attachment 3**, you will find the contents of the `BestFlag.tga` file saved in the specified `.TGA` format, which we viewed in a hex viewer.

Task 1 (X)

Draw which image is stored in the attached `BestFlag.tga` file – i.e. draw a pixel grid (rectangular matrix), and for each saved pixel specify its color (color the pixel square with this color or write a color designation in it: K = black, R = red, G = green, B = blue, W = white). Explain how you came to your result.

Task 2 (X)

For the attached `BestFlag.tga` file, write the date and time when it was saved (all numbers in decimal system). Explain how you came to this result.

Task 3 (X)

Write a Python implementation of the function:

```
def PrintInfo(targaFileName : str):
```

which takes as its argument the name of the .TGA image file. For simplicity, unrealistically assume that only the correct values of the arguments are passed to the function, i.e. there is no need to check their correctness (and therefore also that, for example, the input file is really in the correct format).

The function should display the size of the image in pixels, and the date of its storage (the first line describes the dimensions as width x height, the second date of storage) - the listing should look like this:

```
1024 x 768
4/26/1986
```

If you need to find out the size of a file in bytes, it is possible to get it in Python, for example, as a result of calling a function:

```
os.path.getsize (fileName)
```

Task 4 (X)

We will save the attached `BestFlag.tga` file on a disk formatted with some real file system. Estimate and explain your estimate in detail how many bytes of file data on disk will actually take up (i.e. how much free disk space will be reduced by saving the file).

Task 5

For the representation of signed integers we can use, for example, a representation with an *explicit sign bit*, a *two's complement* or a *bias* representation. Compare these 3 different representations in detail and list their main advantages and disadvantages.
In each of these representations in the 8-bit variant, write the numbers:

```
42
-7
0
```

Task 6

Assume that we want to store the real number `-219.375` to address `0x0C025520` as a single type, i.e. a 32-bit floating-point number according to the IEEE 754 standard, i.e. the mantissa is normalized with a hidden 1 and occupies the lower 23 bits, then followed by 8-bit exponent stored in the [bias] format +127, and the last bit, i.e. **MSb**, **is a sign bit**.
Write in hexadecimal the value of each byte of memory in which some part of this value will be stored - suppose we are working on a 32-bit **little-endian** CPU.

Common part for tasks marked Y

See **attachment 4** for a description of the serial mouse packet format (for RS-232 line) that uses the "Logitech mode" communication protocol.

Task 7 (Y)

Draw a timing diagram of the entire transmission of one mouse packet using the "Logitech mode" communication protocol (bits are sent over the RS-232 serial line in **LSb-first** order) if the mouse sends us the following information in this packet:

- Left button - not pressed
- Middle button - not pressed
- Right button - pressed
- X-axis offset = `-7`
- Y-axis offset = `42`

Task 8 (Y)

Write a Python implementation of the PrintPackets function using standard types from the `numpy` package:

```
def PrintPackets(data : bytes):
```

whose `data` parameter contains bytes received via the RS-232 line from a serial mouse using the "Logitech mode" communication protocol, and in principle contains 1 to N packets received from such a mouse. Your implementation of the function should print one line for each packet contained in the `data` with the following information:

1) the value of the X coordinate in the decimal system
2) the letter L if the left button has been pressed
3) the letter M if the middle button has been pressed.

Think about how you will know if the mouse sent a 3-byte or 4-byte packet (this is the same principle as the mouse from the lecture, by the way).

Common part for tasks marked Z

Assume the 32-bit little-endian CPU described below has a general register architecture based on the x86 architecture (IA-32) – a processor with a 32-bit address space. The processor has, among other things, general registers EAX, EBX, ECX, EDX (for each register there is also a view of its lower 16 bits under the names of registers AX, BX, CX, DX, as well as a view of its lower 8 bits under the names AL, BL, CL, DL), the EFLAGS flag register with common flags, and the EIP (instruction pointer) register. The instruction set includes, among other things, the following instructions (the flag register is modified only by arithmetic operations, but not by data transfer instructions) – each of these instructions has a 32-bit, 16-bit and 8-bit variant, for a specific variant of the instruction both operands are always with the same bit precision (i.e. a 32-bit variant has all 32-bit operands, etc.; we choose the instruction variant by the type of source / target register):

`MOV reg,imm/[addr]`	(load register)
`MOV [addr],reg`	(store register)
`MOV reg0,reg1`	(copy from `reg1` to `reg0`)
`ADD reg,imm/[addr]/reg`	(add without carry)
`ADC reg,imm/[addr]/reg`	(add with carry)
`SUB reg,imm/[addr]/reg`	(subtract without borrow)
`SBB reg,imm/[addr]/reg`	(subtract with borrow)
`CLC`	(clear carry)
`STC`	(set carry)

All the above two-operand instructions always have a **target operand on the left** and a **source operand on the right**.

Instructions can have the following variants of operands for each "bitness" (for allowed variants, see the definitions of a specific instruction):

- immediate value `imm`
- absolute address `[addr]`, where `[addr]` can be:
  - `[imm]` address given by the constant `imm`
  - `[reg32]` address given by the contents of the 32-bit `reg32` register
  - `[reg32 +/- imm]` address given by the sum / difference of the contents of the 32-bit register reg32 and the constant `imm`
- any register `reg`

Task 9 (Z)

Assume that in a C# program we have declared 4 variables a, b, c, d of type `ulong`, which is equivalent to type `uint64` from the `numpy` package. We know that for the variables, the compiler reserved a place immediately after each other in the order a, b, c, d towards the higher addresses, where the first variable a lies at the address `0x00081000`. We also know that for temporary variables we can use any memory between addresses `0x7D000000` and `0x7D00FFFF`. Write in the assembler of the above processor how the following C # expression would be translated (do not perform any optimizations and algebraic simplifications [like removing parentheses]):

```
a = (c + d) – b + 3
```

Task 10 (Z)

Suggest what the machine code of this x86 processor variant might look like - of course, your design does not need to match the real x86 processor architecture, but it should be reasonably realistic and principally match what the machine code of a regular CPU looks like. Assume, of course, that we end up supporting all of the above instructions, however, in your solution, describe in detail only what the machine code will look like for only 32-bit variants of the ADD and SUB instructions. For ADD and SUB instructions, consider all of the above operand variants, however, assume that only one of the above 4 general registers can be used as a register in the instruction operands.

Write the value of each byte of memory where some part of the machine code according to your design would be stored, if we wanted to write the following sequence of instructions in it (we will store the code from the address `0x00000400`):

```
ADD EAX, [12345678h]
ADD EAX, BADDAD00h
ADD ECX, EAX
SUB ECX, EBX
```