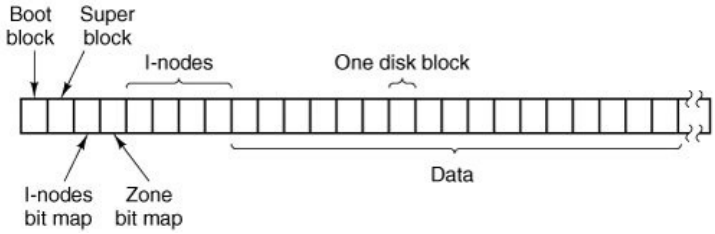
Write your answers to the special response sheet you received (with your name and photograph). If you are using more than a single sheet of paper for your answers, then mark each sheet with its number / total number of sheets you will hand over.

Common part for tasks marked X

Suppose we have a file (see its hexdump / hexview in **attachment 2** – this hexdump uses a special compact convention, where if one or more lines with exactly the same content are repeated after a line of the report, then only 1 line is displayed instead of all repetitions followed by line with an asterisk *), which represents a disk image formatted by the Minix file system – disk image = a file that contains the stored content of all sectors occupied by a file system on another disk; data from sector 0 of the original disk are stored from offset 0, data from sector 1 from offset 512, etc. The Minix filesystem logically divides the disk into so-called blocks or zones, where **1 block has a size of 1 KiB** (i.e. two 512 byte sectors). Block number 0 contains the boot sector, block number 1 is the so-called superblock and contains basic information about the formatted file system. Information about free blocks is stored in a so-called free zone bitmap. Information about one file is stored in a data structure called inode – inodes are numbered from 1. Each file is assigned exactly one inode. The total number of inodes is given when formatting the file system – i.e. this also gives the maximum number of files that we can save to the file system. All inodes are stored one after the other in the inode table (blocks marked I-nodes in the picture below). Information about free inodes is stored in the so-called free inode bitmap (I-nodes bitmap).

Directory files contain a list of pairs - the name of the file and a reference to its inode (inode number). The overall structure of a disk formatted with the Minix file system is as follows:



A more detailed specification of the file system metadata is in **attachment 1** (the incorrectly labeled table in Section 5 describes *superblock*). All file system metadata use the **little-endian** byte order. In the disk image (see **attachment 2**), the inode table starts at offset $1000, the first data block has the number $1A and therefore starts at offset $6800, the root directory data is stored in block $1A (note that the root directory contains two entries "dot" and "two dots", both of which refer to inode number 1, i.e. the inode of the root directory itself).

Task 1 (X)

Write as a number in the decimal system how many total (even unused) inodes are on the disk (number of inodes).

Task 2 (X)

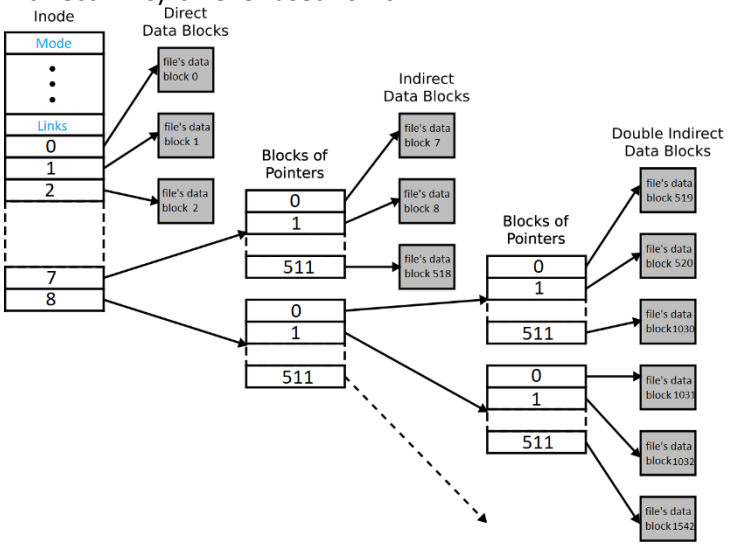
What is the length of the `second.txt` file in bytes? And what is its content? How do you know that its contents are the bytes you mentioned? Explain everything in detail.

Task 3 (X)

Program the following function in Python:

```
def PrintBlocks(name : str, inodeOffset : int):
```

which as the `name` argument gets the name of the disk image file formatted with the Minix filesystem, and in the `inodeOffset` argument the offset from the beginning of the image (image file), where the inode of a file stored in the filesystem is located. The procedure should list the numbers of all blocks (zones) that contain the file data itself (one number per line). Note that the ZONE7 number contains a block number that does not contain the data of the file itself, but only block numbers with the file data itself, similar to ZONE8, but in two levels - see the specifications of part X, and the figure below - however, **in your implementation**, the file, whose inode you are studying, is small enough, and **uses at most ZONE0 to ZONE7** (i.e. **direct links**, and **one-level indirect links**), but ZONE8 (i.e. two-level indirect links) is never used for it:



Task 4 (X)

Write in Python (as needed using standard types and common operations from the numpy package) the implementation of the GetLongestFreeBlock function:

```
def GetLongestFreeBlock(bitmap : bytes) -> int:
```

In the `bitmap` parameter we get the pre-read content of one block with a "bitmap" of free zones – i.e. an object of 1024 bytes. If a certain block (zone) is free, then the bit assigned to it in the bitmap is 0, if the block is occupied, it is in the given bit 1. LSb of the byte 0 is a bit representing the state of block 0, MSb of the byte 0 is bit of block 7, LSb of the byte 1 is the bit of block 8, MSb of the byte 1 is the bit of block 15, etc. Write the GetLongestFreeBlock function to return the block number where the longest continuous sequence of free blocks begins. So for example for data:

`67 F0 F7 FF ... FF`

the function should return 7. **Use bitwise operations for your solution.**

Task 5

The 6502 processor is an **8-bit little-endian** microprocessor with a battery architecture and a **16-bit memory address space**. In addition to the accumulator register (labeled A in the assembler), the processor has two auxiliary registers, X and Y, and a flag register with common flags. The 6502 processor has the following instruction set:

`LDA arg`	Load Accumulator
`STA arg`	Store Accumulator
`SEC`	Set Carry
`CLC`	Clear Carry
`AND arg`	Bitwise AND
`ORA arg`	Bitwise OR
`ASL A`	Shift Left – performs a logical shift of the value in A by one to the left
`ROL A`	Rotate Left – performs a bit rotation of the 9-bit value one to the left – rotates A together with the Carry flag, which serves as the 8th bit of the value (counted from 0) – i.e. the original value of the Carry flag rotates to LSb A, and the original MSb A rotates to the Carry flag as its new value
`TXA, TAX, TYA, TAY, TSX, TXS`	= 6 instructions for copying a value (Transfer) between registers (2nd letter of instruction name = source register, 3rd letter = destination register)

The instruction argument (if they have it) can be one of the following variants - numeric values are implicitly understood in the assembler 6502 in the decimal system, if the value is to be understood in the hexadecimal system, it must be preceded by the prefix `$`:

`#number`	= immediate value
`number`	= memory address
`number,X`	= direct address in memory obtained by calculation – the resulting address in memory is obtained as number + X, i.e. the result of the sum of the value number and the value stored in register X

Arithmetic instructions modify the flag register in the standard way. According to the transferred value, **load and transfer instructions** also modify the flag register (except for the **carry flag**, which they **leave unchanged**).

Assignment: Assume that in a C # program we have declared 3 variables a, b, c of type `ushort`, which is equivalent to type `uint16` from the `numpy` package. Also, assume that standard arithmetic and bitwise operations in C# work the same as in Python on `numpy` types. We know that for the variables, the compiler reserved a place immediately after each other in the order a, b, c towards the higher addresses, where the first variable a lies at address `0xE000`. We also know that for temporary variables we can use any memory between addresses `0x0000` and `0x00FF`. Write in the assembler of the above processor how the following C# command would be translated (do not perform any optimizations and algebraic simplifications):

```
a = ((c & 0x3F07) | b) << 1
```

Task 6

Assume the following fragments of Python programs that use the numpy module. Explain in detail what and why each of these Python code snippets will print:

a)
```
x = 65534
x = ~x
print(x, format(x, "04X"))
```

b)
```
x = 65534
x = ~x
x = uint16(x)
print(x, format(x, "04X"))
```

Task 7

Compare 2 typical implementations of the subtraction: subtract with borrow and subtract with carry. Explain the difference between them and why they are used.

Task 8

Suppose we want to send the text "`MFF`" without quotes in ASCII encoding via the RS-232 line, and we know that the code of the capital letter A is `41h`. We also know that data is sent via RS-232 as LSb-first. Draw a timing diagram of all such transmission, and mark / explain the meaning of each part. Select the required constants appropriately.

Task 9

Assume that we have humidity and temperature sensor type SHT20 from Sensirion, with which we communicate using the standard variant of the I^2C bus (all bytes are sent as MSb-first). See **attachment 3** for the datasheet for this controller (everything important is **on pages 7, 8, and 10**). Write in the hexadecimal system the sequence of all bytes (without ACK bit) that will be transmitted via the I^2C bus, if we want to perform humidity measurements with SHT20, and the sensor returns the measured humidity recalculated as $S_{RH} = 3124_{dec}$. Mark before / after which of the bytes START, resp. STOP condition.

Assume that the measurement from the point of view of communication via the I^2C bus takes place instantly, and there will be no need to wait for the sensor. Also assume that we will not receive a "checksum" as a master, i.e. after receiving the Data LSB master will send a NAK – so you do not need to calculate the "checksum" value.

Task 10

Assume that only zero bytes are stored in memory at addresses `$00AC1400` through `$00AC140F`. The real numbers below will be represented in a fixed-point representation. Then we always store these fixed-point values at the specified address in the mentioned memory in little-endian order. Write the hexdump of the final memory state between addresses `$00AC1400` to `$00AC140F`, after we store `65.75` as a fixed-point `8.24` on `$00AC1404`, and we store `65.75` as a fixed-point `12.20` on `$00AC140A`.