1. Desimal, binary, hexadecimal
   a. What is the 16-bit binary representation of decimal value 2050? Give the result in bits and in hexadecimal notation. Can you just write it down? 0000 1000 0000 0010 = 0x0802
   b. What is the 32-bit binary representation of decimal value 2050? Give the result in bits and in hexadecimal notation. Can you just write it down? 0000 0000 0000 0000 0000 0000 0000 0010 = 0x00000802
   c. What is the hexadecimal presentation of binary 0111010111? What is the decimal value for it? 01 1101 0111 = 0x1D7 = 1 * 2^5 + 13 * 16 + 7 = 256 + 208 + 7 = 471
   d. What is 0x1AB in 32-bit in binary? What is the decimal value for it? 0x1AB = 0x000001AB = 0000 0000 0000 0000 0000 0001 1010 1011 = 256 + 160 + 11 = 427
   e. What is 0x1234ABCDE in binary? What is the decimal value for it? 0x1234ABCDE = 0001 0010 0011 0100 1010 1011 1100 1101 = 305441741

2. Integer representation. Assume that integer value -13 is stored in memory at byte address 0x1230. What is the contents of bytes 0x1230-0x1233, when that number is stored in the following representations:
   a. 32 bit Big-Endian sign and magnitude 0x 80 00 00 0D
   b. 32 bit Big-Endian two's complement 0x FF FF FF F3
   c. 16 bit Big-Endian two's complement 0x FF F3 ?? ??
   d. 8 bit Big-Endian two's complement 0x F3 ?? ?? ??
   e. 16 bit Big-Endian, biased by 32767 (2^15-1) 0x 7F F2 ?? ??
   f. 32 bit Little-Endian two's complement 0x F3 FF FF FF
   g. 8 bit Little-Endian two's complement 0x F3 ?? ?? ??

3. In memory location 0x4320 there is 32-bit value 0x02700064. What value does it denote, if it represents
   a. 32 bit Big-Endian sign & magnitude +0x02700064 = +40894564
   b. 32 bit Little-Endian sign & magnitude +0x64007002 = +167750274
   c. 32 bit Big-Endian floating point? +1.8750119 * 2^-123 = +1.7632527366211459e-37
   d. 32 bit Big-Endian 4 characters with UTF-8 coding? "<ctrl>p<ctrl>d" because 0x02 = <control>, 0x70 = 'p', 0x00 = <control>, 0x64 = 'd'.
   e. 32 bit Big-Endian ttk-91 machine instruction? load r3, @100
   f. 32 bit Big-Endian ARM machine instruction (extra)? RSBCEQ R0,R0,#100 i.e., if EQ (Z), then R0 = 100-R0 (reverse-subtract)

   0x02700064 = 0000 001 0011 1 0000 0000 0000 0110 0100
   EQ  imm RSB B Rn Rd RoR =100

4. Floating point values.
   a. Give an example on a real number that can be represented exactly as a floating point number? 1.0, 4.0, 0.5 -- all powers of 2
   b. Give an example of a real number, that can not be represented exactly a floating point? How large is the error?
      E.g., 1/3, pi
      32 bit IEEE floating point accuracy is one bit more than the number of bits in mantissa, i.e., 23+1 = 24. The integer part of mantissa is hidden, because it is known to be 1.
      So, the possible error is one in 34 million, i.e., one out of 2^25
      This is because the 25th bit is the first "missing" bit in the representation, and it could be anything.
   c. What decimal number is in binary 101101.1010? 45,625
d. What is the IEEE floating point representation for decimal number 5.1?
   Is it exact or not?
   0x 40 A3 33 33, which is not exact, because with greater accuracy all the remaining bits would not be zeroes.

   How about 5.1000007 and 5.1000007?
   0x 40 A3 33 35 ja 0x 40 A3 33 33 (same as 5.1)

e. Compute the expressions "(1.0666668-1.066666) * 1.23456" and "1.0666668*1.23456 - 1.066666*1.23456" in some programming language that supports 32-bit IEEE floating points. Place some dummy output in the middle of the 1st expression before the multiplication (e.g., see test.c), so that the compiler will not optimize your code too much. Why are the results different? Which one is correct? Why would using 64-bit IEEE floating points remove the difference in this case? Will using 64-bit floating points remove the problem completely?
   Mathematically both should be 0,000000246912, which you can obtain by computing by hand. With 32-bit IEEE arithmetic the 1st case will give you 0,000000294342 (19% error) and the 2nd case 0,000000238419 (3.4% error).
   When subtracting two floating point value very close to each other, the result is significantly less accurate than the original values. This is because the leftmost bits cancel each other, and you fill up (random) zeroes to the end at normalization.
   When you use this inaccurate value later on, the error can only multiply.
   If possible, do this type of subtractions at the end of computation, so that the error would not multiply. This is the reason why the latter expression gives better result.

   This problem does not appear as badly with these 9 decimal digit numbers if one would use 64 bit IEEE numbers. However, the problem is always present when subtracting values that are very close to each other. Over time, even samll errors may accumulate.

5. Floating point representation. Notice that the IEEE floating point standard has a special representation for very small numbers.

   a. What is the representation for very small numbers? What is bad with this representation?
      Exponent representation is 0 and its value -126. Hidden bit value is 0.
      E.g., if mantissa representation is 000 0000 0000 0011 1001 0001, its value is 0.000 0000 0000 0011 1001 0001.
      These type of values are not normalised, and the accuracy of the example value is only 8 bits.

   b. What is the smallest positive 32-bit IEEE floating point value? What is its representation? What is its value in decimal?
      Smallest positive mantissa 0.000 0000 0000 0000 0000 0000 0000 0001.
      Its representation is 0 0000 0000 0000 0000 0000 0000 0000 0001.
      Its value is $2^{-23} * 2^{-126} = 2^{-149} = 1.4 * 10^{-45}$

   c. What is the largest positive 32-bit IEEE floating point value? What is its representation? What is its value in decimal?
      Largest exponent is 127 and its representation is 254.
      (Representation 255 is reserved for undefined numbers and infinities).
      Largest mantissa is 1.111 1111 1111 1111 1111 1111.
      Largest number representation is 0 1111 1110 111 1111 1111 1111 1111 1111 1111.
      Its value is $(2 - 2^{-23} ) * 2^{127} = 3.4 * 10^{38}$.

6. Hamming code.
   a. Show, how error correcting Hamming-code locates and corrects the error, when in 7-bit data "011 0100" the 3rd bit from left becomes erroneous: "010 0100".
   These 7 bits include both data and ECC (error correction code).
   Bits are numbered here from right to left. Data has three parity bits: 011 0100.
In erroneous data \textcolor{red}{010 0100} bit 5 is checked by bits 4 and 1. Bit 1 checks bits 1,3,5,7 and bit 4 checks bits 4,5,6,7. They are both wrong, and so bit 5=4+1 (\textcolor{red}{010 0100}) is erroneous. Bit 5 is flipped automatically and now data is coorect: \textcolor{blue}{011 0100}.

b. How many wires (bits) are needed to protect a 32-bit data bus using error correcting Hamming code?
(We want to transfer 32 bits of data in addition to the ECC bits)
Bits 1, 2, 4, 8, 16, and 32 are parity bits.
So, we need altogether 38 bits: dd dddd pddd dddd dddd dddd pddd dddd pddd pdpp.

c. What if it was a 64-bit data bus?
Bits 1, 2, 4, 8, 16, 32 and 64 are parity bits.
We need 7 parity bits and altogether 71 bits.

d. Why is Hamming code not a good solution to protect data transmissions (local area) networks?
In network data transmissions errors happen in large clusters. Hamming code will not guarantee error detection for more than 2 concurrent errors, and there is no chance of automatic error correction. You need some other means of error detection and recovery is done by repeating the whole send operation.

7. Ordinary SEC (Single-Error-Correction) Hamming code will correct 1 bit errors but is does not necessarily even detect 2 bit errors.
   a. Think of a situation where in 7-bit data (4 bit real data and 3 parity bits, even parity)
      "011 0100" two data bits (bits 1 and 2 from the right) change: "011 0100". Show how normal (SEC) Hamming code does not detect this. What does it cause?
      Parity bits 1 and 2 will flip: "011 0111". Now the parity bits for parity classes 1 and 2 are bad, and so Hamming code will flip bit 3: "011 0011". Data is now wrong, but it looks correct because all parity bits are ok.

   b. The situation can be corrected with an extra parity bit (bit 8). How would compute the value for that parity bit? Show the resulting 8-bit data value for the case in part a.
      Parity bit 8 computes even parity for all bits. No other bit protects this new bit 8.
      Data is now in form "\textcolor{red}{1011 0100}".

   c. Show how the 2-bit error in part a is not detected with this SEC-DED Hamming code. (Single-Error-Correction, Double-Error-Detection).
      Errorneous data is now "\textcolor{red}{1011 0111}". hamming code will fix it to form "\textcolor{blue}{1011 0011}". However, now the extra parity bit 8 indicates some error in data.
      In this case with 2 bits erroneously flipping, bit 8 is originally still correct, but after Hamming code flipping bit 3 it (bit 8) will be wrong.

   d. Why does adding one bit help to detect all 2-bit errors? Consider also that one of the errors could be in bit 8.
      Errorneous flipping of 2 bits (not including bit 8) will cause at least one parity error in Hamming code, but parity bit 8 still has correct parity. After repairing the data one of the bits 1-7 is flipped, and then bit 8 will have incorrect parity.
      If bit 8 is one of the erroneously flipped bits, then this special case is recognized immediately. If there is some other parity bit error, you have an unrecoverable situation. If there are no other parity errors, then bit 8 alone is erroneous and can be flipped back.