Started on	Thursday, 5 August 2021, 3:50 PM
State	Finished
Completed on	Friday, 6 August 2021, 4:56 PM
Time taken	1 day 1 hour
Grade	**100.00** out of 100.00

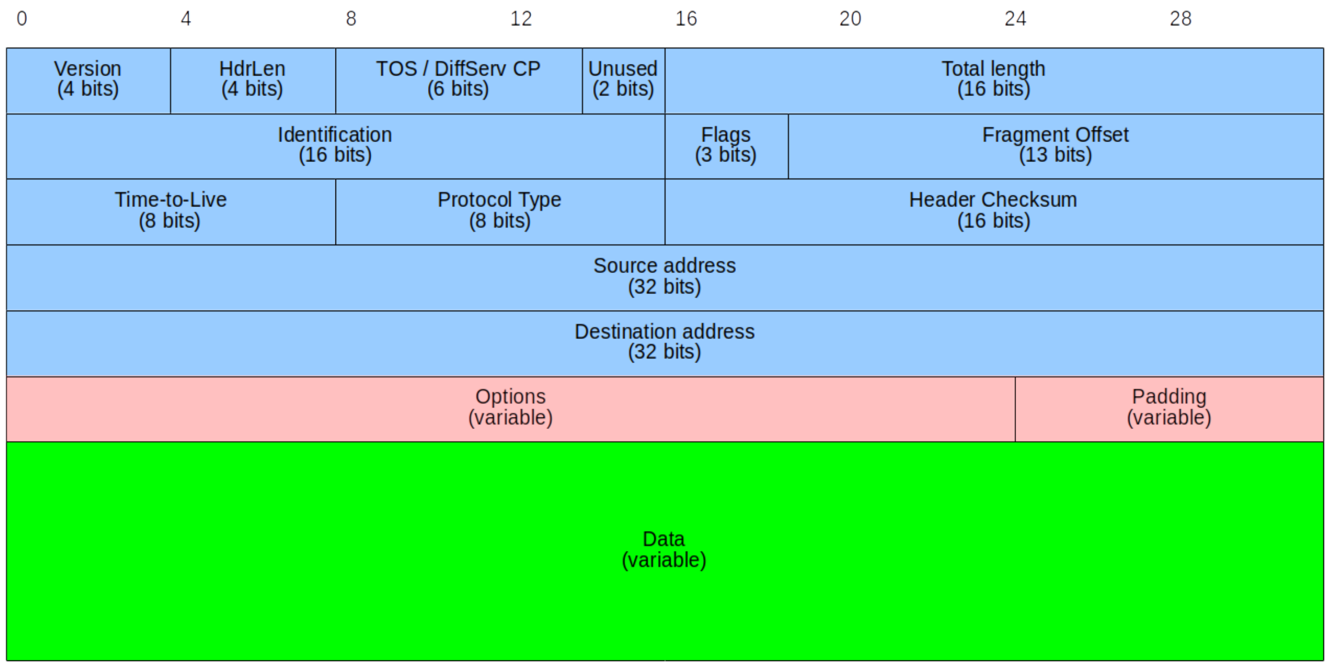
Information

In this superquiz we will look into creating packets for transmission in their proper format, and into checking received packets for validity.

Any existing protocol has very precise rules about the format of the packets it sends or receives. A sender is required to adhere to this format when composing a packet, and a receiver is required to check whether a received packet indeed follows the format, as packets can possibly be changed while being in transit (e.g. due to errors) and we want to avoid processing of erroneous packets. Conceptually, a packet has some similarity to records in programming languages (or in object-oriented languages like Python: the data fields belonging to a class), but additionally there are strict rules about how a packet is represented and stored in memory or while being in transit. The rules for a packet format can for example specify the following things:

- The precise positions and (maximum) sizes of the packet header, payload part and packet trailer (if present).
- Packet headers:
    - Which **fields** make up the header, including a specification of which fields are mandatory and which fields are optional.
    - The precise order in which fields occur in the header.
    - The size of each header field, usually specified as a number of bits.
    - The set of allowed values and their meaning for each header field, and how the field value should be computed.
    - The precise representation/encoding of a header field, e.g. its bit ordering (big-endian versus little-endian).
- Similarly for the packet trailer (if present)
- Maximum payload size (usually measured in bytes)

As an example consider the header of the ubiquitous IP protocol (in IP version 4), which looks as follows:



In this figure the blue part shows the mandatory header fields, the red part shows the optional header fields (which we ignore in this superquiz, i.e. we just assume that the red part is empty) and the green part contains the payload data. One row in this figure corresponds to one chunk of 32 bits. Within memory and on the channel the header fields follow each other in "english reading order": first all the fields from the first row, from left to right, then all the fields from the second row, from left to right, and so on. Within fields larger than one byte, like for example the 'DestinationAddress' field, the bits are stored from the highest-valued bit to the lowest-valued bit, so for example a 32-bit value 0x12345678 is stored as four bytes such that the first byte is 0x12, the second byte is 0x34, the third byte is 0x56 and the last byte is 0x78 -- this "highest-to-lowest" order of storing the bytes belonging to a 32 bit value is called **big endian** order. Note that this is **not** how an Intel 32- or 64-bit processor would store the 32-bit value 0x12345678 in memory -- Intel processors adopt the **low endian** order and store the four bytes in the sequence 0x78, followed by 0x56, then 0x34 and finally 0x12. **Important**: you are asked to store these values in big endian order.

In this superquiz you are asked to write Python programs (more precisely: Python3) allowing you to assemble IPv4 packet headers and extracting certain fields from IPv4 packet headers. You do not need to understand the IPv4 protocol and the meaning of the various fields at this stage. Instead the goal is to familiarize you with typical considerations involved in packet processing, and packet processing is an important component in any communications protocol.

Python has many advantages as a programming language but it is not really built for low-level manipulation of memory areas representing packets -- this is possible but the language can make this a bit clumsy (there is a reason why communication protocols are often implemented in C or C++). Within a Python program the most appropriate way to store a packet in memory is as a bytearray, which is a mutable data structure representing (you guess it) an array of individual bytes. For this superquiz you will need to find out how to work with bytearrays and you will also need to find out how to work with bitwise operations (like the bit shifting operators '<<' and '>>') and bit masking. If you still need a bit of practice here, then have a look at the first week's quiz on "bit fiddling".

Question **1**

Correct

Mark 25.00 out of 25.00

Please complete the following Python3 function which takes as parameters the real values to be filled into the packet, checks these parameters for validity (see below), and which returns either:

- an error code, i.e. an integer specifying a particular error in the parameters handed to the function, or
- a bytearray of 20 bytes length containing the standard IPv4 header (without optional fields), which is only to be returned when all validity checks have passed.

Note that the IPv4 header contains two 'unused' bits, these have to be set to zero.

The following validity checks have to be performed:

- If the value of the 'version' parameter is not the integer 4, then return error code 1
- If the value of the 'hdrlen' parameter does not fit within 4 bits, then return error code 2
- If the value of the 'tosdscp' parameter does not fit within 6 bits then return error code 3
- If the value of the 'totallength' parameter does not fit within 16 bits then return error code 4
- If the value of the 'identification' parameter does not fit within 16 bits then return error code 5
- If the value of the 'flags' parameter does not fit within 3 bits then return error code 6
- If the value of the 'fragmentoffset' parameter does not fit within 13 bits then return error code 7
- If the value of the 'timetolive' parameter does not fit within 8 bits then return error code 8
- If the value of the 'protocoltype' parameter does not fit within 8 bits then return error code 9
- If the value of the 'headerchecksum' parameter does not fit within 16 bits then return error code 10
- If the value of the 'sourceaddress' parameter does not fit within 32 bits then return error code 11
- If the value of the 'destinationaddress' parameter does not fit within 32 bits then return error code 12

In addition to these, all parameters have to be non-negative. You should also assume that all the parameters are integers.

As mentioned above, your function should return a byte-array. When during the development you run your function interactively, you will notice that Python does not print the contents of a byte array consistently: some elements are printed by giving their hexadecimal value (e.g. '\x00'), others are printed as letter or number literals (e.g. 'E' or 'M') when their value falls into one of the ranges of printable characters. This can be a bit confusing.

For example:

Test	Result
`print(composepacket(5,5,0,4000,24200,0,63,22,6,4711, 2190815565, 3232270145))`	`1`
`print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145))`	`bytearray(b'E\x00\x05\xdc^\x88\x00?\x16\x06\x12g\x82\x951M\xc0\xa8\x87A')`

Answer: (penalty regime: 0, 10, 20, ... %)

Reset answer

```python
def composepacket (version, hdrlen, tosdscp, totallength, identification, flags, fragmentoffset, timetolive, protoc

    if version != 4:
        return 1
    if hdrlen.bit_length() > 4 or hdrlen < 0:
        return 2
    if tosdscp.bit_length() > 6 or tosdscp < 0:
        return 3
    if totallength.bit_length() > 16 or totallength < 0:
        return 4
    if identification.bit_length() > 16 or identification < 0:
        return 5
    if flags.bit_length() > 3 or flags < 0:
        return 6
    if fragmentoffset.bit_length() > 13 or fragmentoffset < 0:
        return 7
    if timetolive.bit_length() > 8 or timetolive < 0:
        return 8
    if protocoltype.bit_length() > 8 or protocoltype < 0:
        return 9
    if headerchecksum.bit_length() > 16 or headerchecksum < 0:
        return 10
    if sourceaddress.bit_length() > 32 or sourceaddress < 0:
        return 11
    if destinationaddress.bit_length() > 32 or destinationaddress < 0:
        return 12

    packet = bytearray(20)


    packet[0] = (version << 4) | (hdrlen)

    packet[1] = tosdscp

    packet[2] = (totallength & 0xFF00) >> 8
    packet[3] = totallength & 0x00FF

    packet[4] = (identification & 0xFF00) >> 8
    packet[5] = identification & 0x00FF

    flag_stuff = (flags << 13) | (fragmentoffset)
    packet[6] = (flag_stuff & 0xFF00) >> 8
    packet[7] = flag_stuff & 0x00FF

    packet[8] = timetolive

    packet[9] = protocoltype

    packet[10] = (headerchecksum & 0xFF00) >> 8
    packet[11] = headerchecksum & 0x00FF

    packet[12] = (sourceaddress & 0xFF000000) >> 24
    packet[13] = (sourceaddress & 0x00FF0000) >> 16
    packet[14] = (sourceaddress & 0x0000FF00) >> 8
    packet[15] = sourceaddress & 0x000000FF

    packet[16] = (destinationaddress & 0xFF000000) >> 24
    packet[17] = (destinationaddress & 0x00FF0000) >> 16
    packet[18] = (destinationaddress & 0x0000FF00) >> 8
    packet[19] = destinationaddress & 0x000000FF

    return packet
```

	Test	Expected	Got
✓	`print(composepacket(5,5,0,4000,24200,0,63,22,6,4711, 2190815565, 3232270145))`	1	1
✓	`print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[18])`	135	135
✓	`print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[19])`	65	65
✓	`print(composepacket(4,16,0,4000,24200,0,63,22,6,4711, 2190815565, 3232270145))`	2	2
✓	`print(composepacket(4,15,64,4000,24200,0,63,22,6,4711, 2190815565, 3232270145))`	3	3
✓	`print(composepacket(4,15,63,65536,24200,0,63,22,6,4711, 2190815565, 3232270145))`	4	4
✓	`print(composepacket(4,15,63,65535,65536,0,63,22,6,4711, 2190815565, 3232270145))`	5	5
✓	`print(composepacket(4,15,63,65535,65535,8,63,22,6,4711, 2190815565, 3232270145))`	6	6
✓	`print(composepacket(4,15,63,65535,65535,7,8192,22,6,4711, 2190815565, 3232270145))`	7	7
✓	`print(composepacket(4,15,63,65535,65535,7,8191,256,6,4711, 2190815565, 3232270145))`	8	8
✓	`print(composepacket(4,15,63,65535,65535,7,8191,255,256,4711, 2190815565, 3232270145))`	9	9
✓	`print(composepacket(4,15,63,65535,65535,7,8191,255,255,65536, 2190815565, 3232270145))`	10	10
✓	`print(composepacket(4,15,63,65535,65535,7,8191,255,255,65535, 4294967296, 3232270145))`	11	11

	Test	Expected	Got
✓	print(composepacket(4,15,63,65535,65535,7,8191,255,255,65535, 4294967295, 4294967296))	12	12
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[0])	69	69
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[1])	0	0
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[2])	5	5
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[3])	220	220
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[4])	94	94
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[5])	136	136
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[6])	0	0
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[7])	63	63
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[8])	22	22
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[9])	6	6
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[10])	18	18
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[11])	103	103
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[12])	130	130
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[13])	149	149
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[14])	49	49
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[15])	77	77
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[16])	192	192
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145)[17])	168	168
✓	print(composepacket(4,-3,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145))	2	2
✓	print(composepacket(4,5,0,1500,-24200,0,63,22,6,4711, 2190815565, 3232270145))	5	5
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,-4711, 2190815565, 3232270145))	10	10
✓	print(composepacket(4,5,0,1500,24200,0,63,22,6,4711, 2190815565, 3232270145))	bytearray(b'E\x00\x05\xdc^\x88\x00? \x16\x06\x12g\x82\x951M\xc0\xa8\x87A')	bytearray \x16\x06\

Passed all tests! ✓

Correct

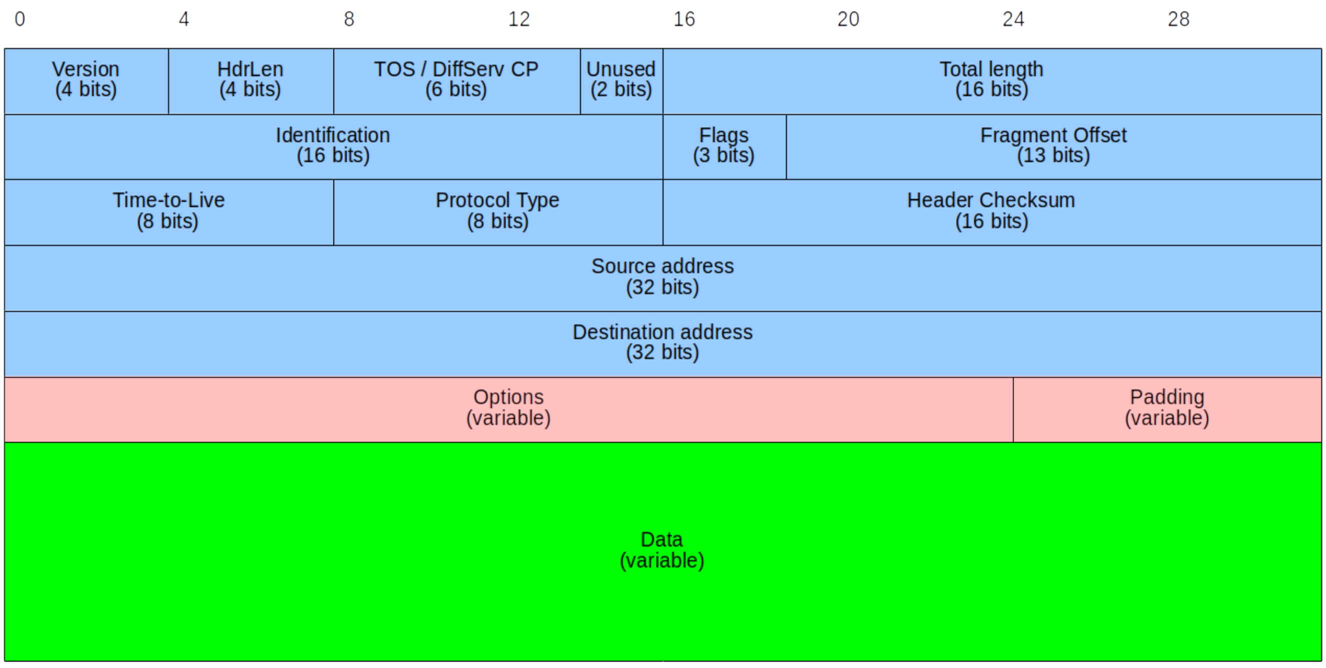
Marks for this submission: 25.00/25.00.

Information

You can do these questions after you have done the "Composing a packet" question.

Suppose that we have received a sequence of bytes which we want to treat as an IPv4 packet. To do this, we first have to run a number of validity checks which ensure that the packet indeed belongs to the IPv4 protocol and is not random garbage (sometimes your communication peers or intermediate routers don't work correctly!), and then we will have to extract various fields from the packet header to make decisions about the packet's further treatment. You are asked to write a sequence of Python functions to help with this.

As a reminder the IPv4 packet format is shown here:



In the following questions you will be handed over a received packet as a Python bytearray.

Question **2**

Correct

Mark 30.00 out of 30.00

In any protocol, the first step after receiving a packet is to carry out basic correctness checks of the packets. In the IPv4 protocol this includes the following things:

1. Check that the packet at least includes a full IPv4 header (i.e. the packet length is at least the minimum length of an IPv4 header).
2. Check that the version number field has the correct value 4.
3. Check that the header checksum is correct ('Header Checksum' field).
4. Check that the total packet length field is consistent with the amount of data you have ('Total length' field).

Please write a Python function which carries out these checks (note that there are further checks, but these are omitted here -- one example is to make sure that the 'Unused' bits in the IPv4 header are 0). The function will be given a single bytearray parameter, which represents the data that has been received.

The 'Total length' field in IPv4 represents the total size of the IPv4 packet, i.e. header plus data.

Your function should return:

- The error code '1' if the packet does not at least include a full IPv4 header.
- The error code '2' if the version field is incorrect.
- The error code '3' if the checksum is incorrect.
- The error code '4' if the total packet length is inconsistent with the amount of data you have.
- The value 'True' if your packet passes all these tests.

Error codes are to be represented as integers. Make sure that these checks are carried out in a sensible order, e.g. that the version field or header checksum is only checked when you are indeed sure that there are enough bytes in the data so that these fields are indeed available.

For the header checksum calculation you can refer to the Wikipedia page https://en.wikipedia.org/wiki/IPv4_header_checksum if you want to understand the background, **but be aware that the method given here is slightly different**. The algorithm for verifying the header checksum works as follows:

- We take the IPv4 header as consisting of ten consecutive 16-bit non-negative integer numbers (with each such 16-bit number stored in big endian format). Add up these ten numbers, call the result X.
- While X is larger than 0xFFFF (i.e. X needs more than 16 bits) do:
  - Let X0 = X & 0xFFFF (i.e. assign to X0 the lowest 16 bits, '&' is bit-wise and).
  - Let X1 = X >> 16 ('>>' is right-shift operator, i.e. shift X to the right by 16 bits)
  - Assign X = X0 + X1
- Check if X is 0xFFFF. If it is, the packet checksum is correct, otherwise the packet checksum is incorrect.

(Note: In this description we have assumed that the header indeed only consists of 20 bytes, i.e. there are no optional header fields. If there were optional header fields, then in the initial step we would have to sum over more as many consecutive 16-bit non-negative integer numbers as are required to cover the entire header including the options. For this problem you will just work with a standard 20-byte header without options.).

Here are three packets for you to test your function while developing it. In all these three cases the packet is entirely correct, so your check function should return 'True' on these.

# pkt1 = bytearray ([0x45, 0x0, 0x0, 0x1e, 0x4, 0xd2, 0x0, 0x0, 0x40, 0x6, 0x20, 0xb4, 0x12, 0x34, 0x56, 0x78, 0x98, 0x76, 0x54, 0x32, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0])
# pkt2 = bytearray ([0x45, 0x0, 0x0, 0x1e, 0x16, 0x2e, 0x0, 0x0, 0x40, 0x6, 0xcd, 0x59, 0x66, 0x66, 0x44, 0x44, 0x98, 0x76, 0x54, 0x32, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0])
# pkt3 = bytearray ([0x45, 0x0, 0x0, 0x1b, 0x12, 0x67, 0x20, 0xe, 0x20, 0x6, 0x35, 0x58, 0x66, 0x66, 0x44, 0x44, 0x55, 0x44, 0x33, 0x22, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0])

For example:

Test	Result
`print(basicpacketcheck(bytearray ([0x45, 0x0, 0x0, 0x1e, 0x4, 0xd2, 0x0, 0x0, 0x40, 0x6, 0x20, 0xb4, 0x12, 0x34, 0x56, 0x78, 0x98, 0x76, 0x54, 0x32, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0])))`	`True`

Answer: (penalty regime: 0, 10, 20, ... %)

Reset answer

```python
def basicpacketcheck (pkt):

    packet_per_header = 4 # bytes per header
    header_length = pkt[0] & 0x0F
    packet_length = packet_per_header * header_length

    if len(pkt) < 20:
        return 1
    elif (pkt[0] >> 4) != 4: # double check this!
        return 2
    elif verify_checksum_correct(pkt, packet_length) == False:
        return 3
    elif pkt[3] != len(pkt):
        return 4
    else:
        return True

def verify_checksum_correct(pkt, packet_length):
    # IPv4 header as consisting of ten consecutive 16-bit non-negative integer numbers
    # Add up these ten numbers, call the result X.

    x = 0
    for i in range(0, packet_length, 2):
        x += int((pkt[i] << 8) | pkt[i+1])

    while x > 0xFFFF:
        x0 = x & 0xFFFF
        x1 = x >> 16
        x = x0 + x1

    return x == 0xFFFF
```

	Test	Expected	Got	
✓	`print(basicpacketcheck(bytearray ([0x45, 0x0, 0x0, 0x1e, 0x4, 0xd2, 0x0, 0x0, 0x40, 0x6, 0x20, 0xb4, 0x12, 0x34, 0x56, 0x78, 0x98, 0x76, 0x54, 0x32, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0])))`	True	True	✓
✓	`print(basicpacketcheck(bytearray ([0x45, 0x0, 0x0, 0x1e, 0x16, 0x2e, 0x0, 0x0, 0x40, 0x6, 0xcd, 0x59, 0x66, 0x66, 0x44, 0x44, 0x98, 0x76, 0x54, 0x32, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0])))`	True	True	✓
✓	`print(basicpacketcheck(bytearray ([0x45, 0x0, 0x0, 0x1b, 0x12, 0x67, 0x20, 0xe, 0x20, 0x6, 0x35, 0x58, 0x66, 0x66, 0x44, 0x44, 0x55, 0x44, 0x33, 0x22, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0])))`	True	True	✓
✓	`print(basicpacketcheck(bytearray ([0x45, 0x0, 0x0, 0x1b, 0x12, 0x67, 0x20, 0xe, 0x20, 0x6, 0x35, 0x58, 0x66, 0x66, 0x44, 0x44, 0x55, 0x44, 0x33, 0x22, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0])))`	4	4	✓
✓	`print(basicpacketcheck(bytearray ([0x45, 0x0, 0x0, 0x1b, 0x12, 0x67, 0x20, 0xe, 0x20, 0x6, 0x35, 0x58, 0x66, 0x66, 0x44, 0x44, 0x55, 0x44, 0x33, 0x22, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0])))`	4	4	✓
✓	`print(basicpacketcheck(bytearray ([0x45, 0x0, 0x0, 0x1b, 0x12, 0x67, 0x20, 0xe, 0x20, 0x6, 0x35, 0x58, 0x66, 0x66])))`	1	1	✓
✓	`print(basicpacketcheck(bytearray ([0x55, 0x0, 0x0, 0x1b, 0x12, 0x67, 0x20, 0xe, 0x20, 0x6, 0x35, 0x58, 0x66, 0x66, 0x44, 0x44, 0x55, 0x44, 0x33, 0x22, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0])))`	2	2	✓
✓	`print(basicpacketcheck(bytearray ([0x46, 0x0, 0x0, 0x1e, 0x16, 0x2e, 0x0, 0x0, 0x40, 0x6, 0xcd, 0x59, 0x66, 0x66, 0x44, 0x44, 0x98, 0x76, 0x54, 0x32, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0])))`	3	3	✓
✓	`print(basicpacketcheck(bytearray ([0x45, 0x0, 0x0, 0x1e, 0x16, 0x2e, 0x0, 0x0, 0x40, 0x6, 0xce, 0x59, 0x66, 0x66, 0x44, 0x44, 0x98, 0x76, 0x54, 0x32, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0])))`	3	3	✓
✓	`print(basicpacketcheck(compose(4, 5, 0, 44, 0x3333, 2, 47, 32, 0x06, 0x44336655, 0x34567890)))`	True	True	✓
✓	`print(basicpacketcheck(compose(4, 5, 0, 44, 0x5555, 2, 47, 35, 0x06, 0x44336655, 0x34567890)))`	True	True	✓
✓	`print(basicpacketcheck(compose(4, 5, 0, 44, 0x3333, 2, 4755, 32, 0x06, 0x44336655, 0x34567890)))`	True	True	✓

Passed all tests! ✓

Correct

Marks for this submission: 30.00/30.00.

Question **3**

Correct

Mark 10.00 out of 10.00

When a received packet has passed the basic sanity tests (version field, header checksum and so on), a host or router needs to extract various header fields for further processing out of the packet.

Please write a Python function which extracts the IPv4 destination address from the received packet (which will be given to you as a bytearray) and which returns a pair (a, dd), where 'a' is the 32-bit value of the destination address, and 'dd' is a string showing the packet address in the so-called "dotted decimal notation", which is ubiquitous for representing IPv4 addresses. You can assume that the received packet has passed all the basic tests (version field, header checksum and so on), you do not need to implement those tests again. To facilitate marking, please make sure that the 'return' statement of your Python function is equivalent to 'return addr, dd' where 'addr' is the 32-bit value of the address and 'dd' is the string with the dotted-decimal notation.

The dotted-decimal notation of a 32-bit IPv4 address is a character string that looks as follows:

aaa.bbb.ccc.ddd

where 'aaa' represents the decimal value of the highest-valued byte of the address, 'bbb' represents the decimal value of the second-highest-valued byte of the address, and so on. For example, if the 32-bit address is 0x12345678 then the dotted decimal output is:

18.52.86.120

For example:

Test	Result
`print(destaddress(bytearray(b'E\x00\x00\x1e\x04\xd2\x00\x00@\x06\x00\x00\x00\x124V3DUf')))`	`(860116326, '51.68.85.102')`

Answer: (penalty regime: 10, 20, ... %)

Reset answer

```python
def destaddress (pkt):

    pkt_address = (pkt[16] << 24) | (pkt[17] << 16) | (pkt[18] << 8) | (pkt[19])
    pkt_dd = str(pkt[16]) + '.' + str(pkt[17]) + '.' + str(pkt[18]) + '.' + str(pkt[19])

    return (pkt_address, pkt_dd)
```

	Test	Expected
✓	`print(destaddress(bytearray(b'E\x00\x00\x1e\x04\xd2\x00\x00@\x06\x00\x00\x00\x124V3DUf')))`	`(8601163` `'51.68.8`
✓	`print(destaddress(bytearray(b'E\x00\x00\x1e\x04\xd2\x00\x00@\x06\x00\x00\x00\x124V\x11"\x88\x99')))`	`(2874758` `'17.34.1`
✓	`print(destaddress(bytearray(b'E\x00\x00\x1e\x04\xd2\x00\x00@\x06\x00\x00\x00\x124V\xa6\xb5\xc4\xb3')))`	`(2796930` `'166.181`

Passed all tests! ✓

Correct

Marks for this submission: 10.00/10.00.

Question **4**

Correct

Mark 10.00 out of 10.00

Now we want to extract the actual payload out of an IPv4 packet. The payload follows the packet header. The length of a packet header is at minimum 20 bytes, but can be longer when options are present. The IPv4 header contains a field 'HeaderLength' which gives the length of the IPv4 header in multiples of 32 bits. So for example, a packet with the minimum header of 20 bytes would carry the value 5 in this field.

Write a Python function which returns the actual payload (and only the payload!) of a packet as a bytearray. You can assume that all the header checks have been successfully carried out before. The packet is given to you as a bytearray.

For example:

Test	Result
`print(payload(bytearray(b'E\x00\x00\x17\x00\x00\x00\x00@\x06i\x8d\x11"3DUfw\x88\x10\x11\x12')))`	`bytearray(b'\x10\x11\`

Answer: (penalty regime: 10, 20, ... %)

Reset answer

```python
def payload (pkt):

    header_length = (pkt[0] & 0xF)

    byte_per_header = 32/8

    header_byte = byte_per_header * header_length

    payload = bytearray()

    i = int(header_byte)
    while i < len(pkt):
        payload.append(pkt[i])
        i += 1

    return payload
```

	Test
✓	`print(payload(bytearray(b'F\x00\x00\x1e\x00\x00\x00\x00@\x06h\x86\x11"3DUfw\x88\x00\x00\x00\x00\x13\x14\x15\x16\`
✓	`print(payload(bytearray(b'E\x00\x00\x17\x00\x00\x00\x00@\x06i\x8d\x11"3DUfw\x88\x10\x11\x12')))`

Passed all tests! ✓

Correct

Marks for this submission: 10.00/10.00.

Question **5**

Correct

Mark 25.00 out of 25.00

Now that you know how to calculate checksums and how to extract payload we consider an updated version of the function to compose a packet.

Please complete the following Python3 function which takes as parameters the actual values to be filled into the packet header (all of these are integer values) and also a bytearray containing the packet payload. Your function should check these parameters for validity as before (see below), and return either:

- an error code, i.e. an integer specifying a particular error in the parameters handed to the function, or
- a bytearray containing the entire IPv4 packet (header, payload), which is only to be returned when all validity checks have passed. When an extended header is required (i.e. when the 'hdrlen' parameter is greater than 5) then the additional 32-bit words making up the header options should be filled with zero bytes.

Note that the IPv4 header contains two 'unused' bits, these have to be set to zero.

The following validity checks have to be performed:

- If the value of the 'hdrlen' parameter does not fit within 4 bits or is smaller than five (minimum header length), then return error code 2
- If the value of the 'tosdscp' parameter does not fit within 6 bits then return error code 3
- If the value of the 'identification' parameter does not fit within 16 bits then return error code 5
- If the value of the 'flags' parameter does not fit within 3 bits then return error code 6
- If the value of the 'fragmentoffset' parameter does not fit within 13 bits then return error code 7
- If the value of the 'timetolive' parameter does not fit within 8 bits then return error code 8
- If the value of the 'protocoltype' parameter does not fit within 8 bits then return error code 9
- If the value of the 'headerchecksum' parameter does not fit within 16 bits then return error code 10
- If the value of the 'sourceaddress' parameter does not fit within 32 bits then return error code 11
- If the value of the 'destinationaddress' parameter does not fit within 32 bits then return error code 12

These checks are identical to the checks in your previous composition function, only the treatment of the 'hdrlen' parameter changes. In addition to these, all parameters have to be non-negative. You will notice that this function has not the same parameters as the previous composition function you have developed. In particular:

- The 'version' parameter is missing, you will need to set it to 4.
- The 'totallength' parameter is missing, you will need to calculate this yourself from the header length (note that the length of the header in bytes is given by the value of the 'headerlength' parameter times four) and the payload length.
- The 'headerchecksum' parameter is missing, as you will need to calculate this yourself. The algorithm for this computation is very similar to the algorithm used when checking a received packet:
  - First fill in all the other fields of the IPv4 header. Initialize the 'headerchecksum' field of the IPv4 header with zero.
  - We take the IPv4 header as consisting of *N* consecutive 16-bit non-negative integer numbers (with each such 16-bit number stored in big endian format), where *N* is the total length of the header in bytes divided by two (or equivalently: *N* is equal to the value of the 'hdrlen' parameter times two). Add up these *N* numbers, call the result X.
  - While X is larger than 0xFFFF (i.e. X needs more than 16 bits) do:
    - Let X0 = X & 0xFFFF (i.e. assign to X0 the lowest 16 bits, '&' is bit-wise and).
    - Let X1 = X >> 16 ('>>' is right-shift operator, i.e. shift X to the right by 16 bits)
    - Assign X = X0 + X1
  - Invert every bit of X, call the result again X
  - Write the value of X into the 16-bit 'HeaderChecksum' header field (and be respectful of the right byte-ordering).
- There is a new parameter called 'payload'. This is a bytearray containing the actual user data within the IPv4 packet. The payload length can be zero or positive. You do not need to check that this parameter is of the right type.

You can re-use and extend the code you have developed for the original 'compose' function, the error codes for all the parameters in common to both versions are the same.

Note: since the case 'hdrlen' > 5 is allowed for your function, the header may actually be longer than 20 bytes. Please simply initialize these optional header bytes with zero.

For example:

Test	Result
print(revisedcompose (6, 24, 4711, 0, 22, 64, 0x06, 0x22334455, 0x66778899, bytearray([0x10, 0x11, 0x12, 0x13, 0x14, 0x15])))	bytearray(b'F`\x00\x1e\x12g\x00\x16@\x06\x11e"3DUfw\x88\x99\x00\x00\x00\x00\x10\x11\x12\x13\x14

Answer: (penalty regime: 0, 10, 20, ... %)

Reset answer

```python
def composepacket (packet, header_bytes, version, hdrlen, tosdscp, totallength, identification, flags, fragmentoff

    # if version != 4:
        # return 1
    if hdrlen.bit_length() > 4 or hdrlen < 5:
        return 2
    if tosdscp.bit_length() > 6 or tosdscp < 0:
        return 3
    # if totallength.bit_length() > 16 or totallength < 0:
        # return 4
    if identification.bit_length() > 16 or identification < 0:
        return 5
    if flags.bit_length() > 3 or flags < 0:
        return 6
    if fragmentoffset.bit_length() > 13 or fragmentoffset < 0:
        return 7
    if timetolive.bit_length() > 8 or timetolive < 0:
        return 8
    if protocoltype.bit_length() > 8 or protocoltype < 0:
        return 9
    if headerchecksum.bit_length() > 16 or headerchecksum < 0:
        return 10
    if sourceaddress.bit_length() > 32 or sourceaddress < 0:
        return 11
    if destinationaddress.bit_length() > 32 or destinationaddress < 0:
        return 12

    packet[0] = (version << 4) | (hdrlen)

    packet[1] = tosdscp << 2

    packet[2] = (totallength & 0xFF00) >> 8
    packet[3] = totallength & 0x00FF

    packet[4] = (identification & 0xFF00) >> 8
    packet[5] = identification & 0x00FF

    flag_stuff = (flags << 13) | (fragmentoffset)
    packet[6] = (flag_stuff & 0xFF00) >> 8
    packet[7] = flag_stuff & 0x00FF

    packet[8] = timetolive

    packet[9] = protocoltype

    packet[10] = (headerchecksum & 0xFF00) >> 8
    packet[11] = headerchecksum & 0x00FF

    packet[12] = (sourceaddress & 0xFF000000) >> 24
    packet[13] = (sourceaddress & 0x00FF0000) >> 16
    packet[14] = (sourceaddress & 0x0000FF00) >> 8
    packet[15] = sourceaddress & 0x000000FF

    packet[16] = (destinationaddress & 0xFF000000) >> 24
    packet[17] = (destinationaddress & 0x00FF0000) >> 16
    packet[18] = (destinationaddress & 0x0000FF00) >> 8
    packet[19] = destinationaddress & 0x000000FF

    headerchecksum = calculate_checksum(header_bytes, headerchecksum, packet)

    packet[10] = (headerchecksum & 0xFF00) >> 8
    packet[11] = headerchecksum & 0x00FF

    return packet

def calculate_checksum(header_bytes, headerchecksum, pkt):
    # IPv4 header as consisting of ten consecutive 16-bit non-negative integer numbers
    # Add up these ten numbers, call the result X.

    n = header_bytes/2

    x = 0
    for i in range(0, header_bytes, 2):
        x += int((pkt[i] << 8) | pkt[i+1])

    while x > 0xFFFF:
        x0 = x & 0xFFFF
        x1 = x >> 16
        x = x0 + x1

    x = ~x

    return x

def revisedcompose (hdrlen, tosdscp, identification, flags, fragmentoffset, timetolive, protocoltype, sourceaddres

    version = 4

    bytes_per_header = 4 # packet per header
    header_bytes = bytes_per_header * hdrlen # packet_length

    totallength = header_bytes + len(payload)

    optional_header = bytearray(header_bytes)

    headerchecksum = 0

    pkt = composepacket(optional_header, header_bytes, version, hdrlen, tosdscp, totallength, identification, flag

    for i in range(len(payload)):
        pkt.append(payload[i])

    return pkt
```

	Test	Expected
✓	print(revisedcompose (6, 24, 4711, 0, 22, 64, 0x06, 0x22334455, 0x66778899, bytearray([0x10, 0x11, 0x12, 0x13, 0x14, 0x15])))	bytearray(b'F`\x00\x1e\x12g\x00\x16@\x06\x11e"3DUfw\x88\x99\x0
✓	print(revisedcompose(16,0,4000,0,63,22,0x06, 2190815565, 3232270145, bytearray([])))	2
✓	print(revisedcompose(4,0,4000,0,63,22,0x06, 2190815565, 3232270145, bytearray([])))	2
✓	print(revisedcompose(5,64,4000,0,63,22,0x06, 2190815565, 3232270145, bytearray([])))	3
✓	print(revisedcompose(5,63,0x10000,0,63,22,0x06, 2190815565, 3232270145, bytearray([])))	5
✓	print(revisedcompose(5,63,4711,8,63,22,0x06, 2190815565, 3232270145, bytearray([])))	6
✓	print(revisedcompose(5,63,4711,0,8192,22,0x06, 2190815565, 3232270145, bytearray([])))	7
✓	print(revisedcompose(5,63,4711,0,8191,256,0x06, 2190815565, 3232270145, bytearray([])))	8
✓	print(revisedcompose(5,63,4711,0,8191,64,256, 2190815565, 3232270145, bytearray([])))	9
✓	print(revisedcompose(5,63,4711,0,8191,64,0x06, 4294967296, 3232270145, bytearray([])))	11
✓	print(revisedcompose(5,63,4711,0,8191,64,0x06, 2190815565, 4294967296, bytearray([])))	12
✓	print(revisedcompose (5, 24, 4711, 0, 22, 64, 0x06, 0x22334455, 0x66778899, bytearray([0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17])))	bytearray(b'E`\x00\x1c\x12g\x00\x16@\x06\x12g"3DUfw\x88\x99\x1
✓	print(revisedcompose (5, 24, 4711, 0, 22, 64, 0x06, 0x66778899, 0x22334455, bytearray([0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17])))	bytearray(b'E`\x00\x1c\x12g\x00\x16@\x06\x12gfw\x88\x99"3DU\x1

Passed all tests! ✓

Correct

Marks for this submission: 25.00/25.00.

◄ Quiz: Circuit- and packet switching (Practice copy)

Jump to...

Superquiz: Packet processing with Python (Practice copy) ►