Question 4:

a. We want to store a binary encoding of the 150 original Pokemon. How many bits do we need to use?

8 bits (a byte)

$2^7 = 128 < 150 < 256 = 2^8$

b. What is the encoding for Pikachu (#25)?

$25 = 16 + 8 + 1 = 1*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 =$ 0b11001. Because we are using an encoding for all 150 Pokemon, the encoding needs to be 8 bits long, so the answer is 0b00011001.

Question 2: *Flippin' Fo' Fun* (10 points, 14 minutes)

Assume that the most significant bit (MSB) of x is a 0. We store the result of flipping x's bits into y. Interpreted in the following number representations, how large is the magnitude of y relative to the magnitude of x? Circle ONE choice per row. **(2 pts each)**

Unsigned	$\|y\| < \|x\|$	$\|y\| = \|x\|$	$\|y\| > \|x\|$	Can't Tell
One's Complement	$\|y\| < \|x\|$	$\|y\| = \|x\|$	$\|y\| > \|x\|$	Can't Tell
Two's Complement	$\|y\| < \|x\|$	$\|y\| = \|x\|$	$\|y\| > \|x\|$	Can't Tell
Sign and Magnitude	$\|y\| < \|x\|$	$\|y\| = \|x\|$	$\|y\| > \|x\|$	Can't Tell
Biased Notation (e.g. FP exponent)	$\|y\| < \|x\|$	$\|y\| = \|x\|$	$\|y\| > \|x\|$	Can't Tell

- In unsigned, a number with the MSB of 1 is always greater than one with a MSB of 0.
- In one's complement, flipping all of the bits is the negation procedure, so the magnitude will be the same.
- In two's complement, y is a negative number. Its magnitude can be found by applying the negation procedure, which is flipping the bits and then adding 1, resulting in a larger magnitude than x.
- In sign and magnitude, the 2^{nd} MSB bit will determine the relative magnitudes of x and y, so you can't tell for certain.
- In biased notation, you read the number the same as unsigned but apply a constant bias to BOTH numbers, so the relation is the same as in unsigned numbers.

Question 1: *Reppin' Yo Numbas* (10 points, 16 minutes)

For this question, we are using 16-bit numerals. For Floating Point, use 1 sign bit, 5 exponent bits, and 10 mantissa bits. For Biased use a bias of $-2^{15}+1$.

Scoring: +1pt for every correctly filled blank, -0.5 for additional incorrect responses in part (a), minimum of 0.

a) Indicate in which representation(s) the numeral is **closest to zero**: Two's Complement (T), Floating Point (F), or Biased (B). The first one has been done for you. NaN is not valid in comparisons.

Numeral:	Closest to zero:
1) `0x0000`	___TF___
2) `0xFFFF`	___**T**____
3) `0x0001`	___**F**____
4) `0xFFFE`	___**T**____
5) `0x8000`	___**F**____
6) `0x7FFF`	___**B**____

b) We now wish to **add** the numerals from **top to bottom (1 to 6)**. However, it is possible that we encounter an error when performing these addition operations. For each number representation, state the FIRST error that is encountered and which numeral causes it; if no error is encountered, answer "no error."

Possible arithmetic errors are: OVERFLOW, UNDERFLOW, NaN, and ROUNDING (assume we are rounding using a truncating scheme).

Representation:	Arithmetic Error:	Numeral #:
Two's Complement	_____**Overflow**_______	__**5**__
Floating Point	______**NaN**__________	__**2**__
Biased	_____**No Error**________	__**x**__ **(this blank was worth zero points)**

Question 2: *This Problem is Like a Box of Chocolates* (27 points, 48 minutes)

a) 4 points. In C, `char` is in fact a *signed* variable type. Assume we have 4 `char`s != 0x00 loaded into a 32-bit `int`, one in each byte. Complete the function below that will negate the `char` in byte `i` of the `int` "in place", with byte 0 being the least significant and byte 3 being the most significant.

```
void negByte(int *data, char i) {

    *(((char *)data)+3-i) = -*(((char *)data)+3-i);

Also accepted: *data = *data^(0xFF << (8*i)) + (1 << (8*i))

}
```

Parts of the problem:
+1 point for dereferencing data and storing the value back
+1 point for manipulating the memory location of the i-th byte within data
+2 points for correctly negating the byte. This meant both flipping the bits and adding one if students took the masking approach.
Because of the wide range of responses, scoring was done holistically. Deductions of 0.5 points were given for small mistakes (eg. Issues with parentheses).

Question 1: Number Representation (15 pts)

a) Complete the tables below: (6 pts)

Convert unsigned integers: (4 pts)

Base 8	**Hexadecimal**
115_8	0x4D
32_8	0x1A

Convert to and from IEC prefixes: (2 pts)

Standard	**IEC Prefixes**
2^{54} bits	16 Pebi-bits
2048 students	2 Kibi students

b) Due to limitations in storage space, we are using only 4 bits to represent integers.

1) What is the *most negative 2's complement* signed integer (decimal) we can represent? (1 pt)

$-2^{n-1} = -2^3 = -8$

2) What is the value (decimal) of the 2's complement number `0b1010`? (1 pt)

-6

3) Write a number (binary) that, when added to `0b0100`, will cause *signed overflow*. (2 pts)

`0b0100 > ANS > 0b0111`

c) An amino acid is defined by a set of 3 consecutive nucleotides (A, C, G, or T). For example, ATG is Methionine. All combinations are unique (e.g. ATG ≠ AGT ≠ GTA).

1) How many total *possible* amino acids are there? (1 pt)

$4^3 = 64$ amino acids

2) In reality, there are 21 amino acids found in the human body. How many bits would it take to encode these amino acids in binary? (1 pt)

$\text{ceil}(\log_2(21)) = 5$ bits

3) Scientists also use single-digit encodings for amino acids (e.g. 'A' for Alanine). In a single sentence, explain why it is okay that we use A for the amino acid Alanine, the nucleotide adanine, and the hex representation of the decimal number 10. (1 pt)

They are different representation systems. We need to know the appropriate interpretation to distinguish the values for our application.

4) We wish to encode the 21 amino acids in base 2, 3, or 5. Which of these choices allows for the MOST new amino acids discoveries before needing to increase the number of digits and how many new discoveries are allowed in this choice? (2 pts)

Base: 2	Possible New Discoveries: 32 – 21 = 11

Question 2: C Potpourri (12 pts)

a) Given the library function `rand()` that returns a random number between `0` and `(2^32)-1` when called, write a valid C expression that uses *bit operations* `(^, ~, |, &)` to initialize the variable `r` with a random integer between `0` and `n`, which is some power of 2 less than `(2^32)-1`. (3 pts)

```
int n = 8; // In this case, we want r to contain one of {0,1,2,3,4,5,6,7}.

int r = rand() & (n - 1);
```