1. What is the value of y after both of the following operations?

```
x = x ^ (~y);
y = y ^ x;
```

~x

```
y = y ^ (x ^ (~y)) → (y ^ ~y) ^ x → 1s ^ x == ~x
```

```
After you plug in x, you can use the commutative and
associative properties of XOR and do y^~y first which results
in all 1s. x XORed with 1s flips its bits, thus ~x
```

Say x = 0111 and y is 1010
0111 ^ 0101 = 0010
1010^0010 = 1000 which is ~x

2. Given the following declarations, do the statements below always evaluate to true?

```
int x = foo();
int y = bar();
unsigned ux = cookie();
```

a.

```
x > ux   ====> (~x+1)  < 0
```

FALSE
Consider x = -1.

- The binary is all 1s, thus when you do ~(all 1s) it becomes all 0s.
    - Adding the 1 makes the value positive.

This is true for all negative x values since the sign bit will always be flipped to 0.

- So the ‘it follows’ is not true for all x > ux.

b.

```
ux - 2 >= -2 ====> ux <= 1
```

TRUE
If ux is 0

- it is comparing the unsigned values of -2 and -2, which are equal.

If ux is 1

- it is comparing the unsigned values of -1 and -2, which are Umax vs Umax -1.

2,3, etc

- aren't true and ux can not be a negative value.

So, it follows that ux must be 0 or 1.

c.
```
(x^y)^x == (x+y)^((x+y)^y)
```
TRUE

Notice that both sides are of the form (A^y)^A

- For the left hand side, A = x
- For the right hand side, A = x+y

(A^y)^A is equivalent to y

- Thus, the equivalence simplifies to y == y
- Both sides of the equivalence are equal

d.
```
(x < 0) && (y < 0) == (x + y) < 0
```
FALSE

Say x == INTMin and y == INTMin.

- (x+y) would overflow.

3. `char** apple[5][9];` 360 bytes (8 * 5 * 9)
   `char* banana[1][9];` 72 bytes (8 * 1 * 9)
   `char strawberry[4][2];` 8 bytes (1 * 4 * 2)

   How many bytes of space would these declarations require?

4. Consider the following struct:
```
typedef struct {
    char first;
    int second;
    short third;
} stuff;
```

Say we are debugging an application in execution using gdb on a 64-bit, little-endian architecture. The application has a variable called array - defined as:
`stuff array[2][2];`

Using gdb we find the following information at a particular stage in the application:

```
[(gdb) p &array
 $1 = (stuff (*)[2][2]) 0x7fffffffe020
```

And:

```
[(gdb) x/48xb 0x7fffffffe020
 0x7fffffffe020: 0x61    0x00    0x00    0x00    0x08    0x00    0x00    0x00
 0x7fffffffe028: 0x02    0x00    0x00    0x00    0x62    0x00    0x00    0x00
 0x7fffffffe030: 0x64    0x00    0x00    0x00    0x04    0x00    0x00    0x00
 0x7fffffffe038: 0x63    0x04    0x40    0x00    0xed    0x03    0x00    0x00
 0x7fffffffe040: 0xc8    0x00    0xff    0xff    0x64    0x7f    0x00    0x00
 0x7fffffffe048: 0x17    0xa6    0x00    0x00    0xe1    0x00    0x00    0x00
```

What is the value of
`array[1][0].second`

At this particular stage of the application?
i.e. what would be returned from the statement:

```
printf("%d\n", array[1][0].second);
```

1005
Because of alignment, each object of type "`stuff`" is 12 bytes.
Due to how arrays are stored in memory,

- The array is stored as:
  array[0][0], array[0][1], array[1][0], array[1][1]

From the gdb output, we can tell that the array starts at `0x7fffffffe020`

- array[1][0] is `0x7fffffffe038` to `0x7fffffffe043`
  - Note: this is in hex, so `0x7fffffffe038` + 8 = `0x7fffffffe040`

`Second` is an integer, and is the 5th to 8th byte of an object of type "`stuff`"

- These are bytes `0x7fffffffe03c` to `0x7fffffffe03f`
- They have the values `0xed`, `0x03`, `0x00`, `0x00`
- Since this system is little endian, the value is `0x000003ed`
  - This is equivalent to 1005

5. The following is part of the result of the command '`objdump -d`' on an executable

```
00000000004006dd <IronMan>:
  4006dd:       55                      push   %rbp
  4006de:       48 89 e5                mov    %rsp,%rbp
  4006e1:       89 7d ec                mov    %edi,-0x14(%rbp)
  4006e4:       8b 45 ec                mov    -0x14(%rbp),%eax
  4006e7:       c1 e0 04                shl    $0x4,%eax
  4006ea:       89 45 fc                mov    %eax,-0x4(%rbp)
  4006ed:       8b 45 fc                mov    -0x4(%rbp),%eax
  4006f0:       5d                      pop    %rbp
  4006f1:       c3                      retq
```

Say the declaration for the function IronMan was:

`int IronMan(int scraps);`

Given that the integer 23 was passed into the function, what is the return value?

368

After instructions 0x4006e1 and 0x4006e4, the input (which was stored in %rdi) is now stored in %eax

Instructions 0x4006e7 then shifts %eax to the left by 4

- This is equivalent to multiply by 2^4, which is 16

23 * 16 = 368

6. The following is a continuation from the previous problem:

```
0000000000400721 <Hulk>:
  400721:       55                      push   %rbp
  400722:       48 89 e5                mov    %rsp,%rbp
  400725:       48 83 ec 20             sub    $0x20,%rsp
  400729:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
  40072d:       48 8b 45 e8             mov    -0x18(%rbp),%rax
  400731:       48 89 c7                mov    %rax,%rdi
  400734:       e8 27 fe ff ff          callq  400560 <atoi@plt>
  400739:       89 45 fc                mov    %eax,-0x4(%rbp)
  40073c:       8b 45 fc                mov    -0x4(%rbp),%eax
  40073f:       89 c7                   mov    %eax,%edi
  400741:       e8 97 ff ff ff          callq  4006dd <IronMan>
  400746:       89 45 f8                mov    %eax,-0x8(%rbp)
  400749:       81 7d f8 8f 01 00 00    cmpl   $0x18f,-0x8(%rbp)
  400750:       7e 10                   jle    400762 <Hulk+0x41>
  400752:       81 7d f8 f4 01 00 00    cmpl   $0x1f4,-0x8(%rbp)
  400759:       7f 07                   jg     400762 <Hulk+0x41>
  40075b:       b8 01 00 00 00          mov    $0x1,%eax
  400760:       eb 05                   jmp    400767 <Hulk+0x46>
  400762:       b8 00 00 00 00          mov    $0x0,%eax
  400767:       c9                      leaveq
  400768:       c3                      retq
```

Given that the function returns 1, what do we know about the value of %edi right before instruction 0x400741 is executed?

%edi is between 25 and 31

Since the function returns 1, we know that the jump instructions at 0x400750 and 0x400759 did not jump.

- From instructions 0x400749 and 0x400750
  - we know that we would have jumped if -0x8(%rbp) was less than or equal to 0x18f
  - Thus we know -0x8(%rbp) is greater than 0x18f, or 399
- From instructions 0x400752 and 0x400759
  - We know that we would have jumped if -0x8(%rbp) was greater than 0x1f4
  - Thus we know -0x8(%rbp) is less than or equal to 0x1f4, or 500
- Thus we know that -0x8(%rbp) is between 400 and 500, inclusive
  - Thus %eax is between 400 and 500, inclusive

From the previous question, we know that IronMan multiplies inputs by 16

- We also know that the function returns a value between 400 and 500 with input %rdi
- Reversing the function, we know the input must have been between 400/16 and 500/16

Thus we know that %rdi was between 25 and 31 right before the IronMan function call

7. Convert the 8-bit floating point number e7 (in hex) to decimal.

A. Convert: $e7_{16} = 11100111_2$.

B. Seprate: $\boxed{1}\boxed{110}\boxed{0111}$

C. Mantissa: 1.0111

D. Exponent: $110_2 = 6_{10}$; $6 - 3 = 3$.

E. De-normalize: $1.0111_2 \times 2^3 = 1011.1$

F. Convert:

Exponents	2^3		2^2		2^1		2^0		2^{-1}		
Place Values	8		4		2		1		0.5		
Bits	1		0		1		1	.	1		
Value	8			+	2	+	1	+	0.5	=	11.5

G. Sign: negative.

Result: e7 is -11.5

b) Convert the number -15.5 into floating point, using 1 sign bit, 3 exponent bits, and 4 mantissa bits.

-15.5 = -1111.1 (-15 + 1^-1)
-1111.1 = -1.1111 x 2^3 (where -1.1111 is base 2)
We have 3 exponent bits, so we calculate our bias as 2^(3 - 1) - 1 = 3

Our sign bit is 1, because out number is negative.
Our exponent bits are 110, because we have an exponent of 3, and a bias of 3.
Our mantissa bits are 1111, because that is our decimal after the .

Our final result:
1110 1111