1.
How many bytes would the following data structures require?

```
struct ucla {
      char blue[6];
      union {
            int gold;
            char joe[8];
      } bruin;

} arr[4];
```

The char array requires 6 bytes. The union requires the number of bytes of its largest data type. In this case, the union requires 8 bytes. In order for the union to be correctly aligned, there needs to be 2 bytes of padding after the first char array. The struct has a size of 16 bytes. There are 4 instances of this struct in the array arr, so in total we need 64 bytes.

2.
What do I need to do if I want to access a function through buffer overflow?
The function’s address is 0x500142.

The function Gets is similar to the standard library function gets—it reads a string from standard input (terminated by ‘\n’ or end-of-file) and stores it (along with a null terminator) at the specified destination (such as a char array previously declared). Functions Gets() and gets() have no way to determine whether their destination buffers are large enough to store the string they read.

```
Dump of assembler code for function getbuf:
=> 0x0000000000601748 <+0>:       push   %rax
   0x000000000060174c <+4>:       sub    $0x40,%rsp
   0x000000000060174f <+7>:       mov    %rsp,%rdi
   0x0000000000601754 <+12>:      callq  0x40198a <Gets>
   0x0000000000601759 <+17>:      add    $0x40,%rsp
   0x000000000060175d <+21>:      pop    %rax
   0x000000000060175f <+23>:      retq
```

Have 64 bytes of padding for the sub and 8 bytes to account for the push (draw the stack). When the function returns, we want the address popped into %rip to be the address of the function we want to run. Thus, we need 72 bytes of padding, and we place the address after that. This is the hex input we would pass in to a magical function like hex2raw which would give us the raw string that when fed into getbuf and placed onto the stack, would convert to the hex input.

```
PP AA DD DD II NN GG 00
PP AA DD DD II NN GG 00
PP AA DD DD II NN GG 00
PP AA DD DD II NN GG 00
PP AA DD DD II NN GG 00
PP AA DD DD II NN GG 00
PP AA DD DD II NN GG 00
PP AA DD DD II NN GG 00
PP AA DD DD II NN GG 00
42 01 50 00 00 00 00 00
```

3.

What is the value of the following 8-bit, tiny floating point number?

`01100000`

Answer:

32

S = 0

E = 12 - 7 = 5

M = 1 + 0 = 1

$(-1)^0 * (1.000) * 2^{12 - 7} = 1 * 1 * 32 = 32$

What about the single precision 32-bit floating point number?

`01000010000001000000000000000000  or  0x42040000`

Answer:

33

S = 0

E = 132 - 127 = 5

$M = 1 + 2^{-5} = 1.03125$

$(-1)^0 * (1.03125) * 2^{132 - 127} = 1 * 1.03125 * 32 = 33$

What is -13.141 in single precision 32-bit floating point format?

Answer:

11000001010100100100000000000000 or **0xC1524000**

Recall that the formula for converting floating point into binary is: Value = $(-1)^S *M* 2^E$

...where E = Exp - Bias

We can convert the magnitude of the input into binary

$.141 = .001001_2$

$13 = 1101_2$

$13.141 = 1101.001001_2$

$= 1.101001001_2*2^3$

We know the input has 3 parts: the signed bit, 8 bits for the Exp field, and 23 bits for the fractional part

11000001010100100100000000000000

- S = 1_2 (because the number is negative)
- Exp = 10000010_2, which is 130 in decimal (because Bias is 127 for single precision, and we want a power E of 3, and Exp - Bias = E = 130 - 127 = 3)
- Frac = $10100100100000000000000_2$ (this is just the mantissa of the original number, $1.101001001_2 * 2^3$ - we use the whole number without the leading 1, which is implied)

4. Designing a (Better?) Floating Point

Hopefully this leads to a better understanding of why IEEE floating point is the way it is

What do we want to represent with floating point numbers?

Represent non-integer values. We want real numbers!

Is it possible to represent all of the numbers we would like to represent?

No :(We only have a finite amount of memory

How can we deal with the above?

With approximation: bounded, but with large (and small!) values

What qualities do we want from our representation?

Open ended, but some examples are simplicity and efficiency

So approximation (and hence precision) is going to play a large role in the design of our floating point numbers. How can we build in the idea of precision into our numbers? (We would like our numbers to be as precise as possible. Think about how errors build when we perform arithmetic operations - maybe what you learned about significant figures will help)

What are some problems with the representation you came up with, and how can they be addressed?

Some possible issues to consider:
Range of numbers?
Precision of numbers?
Overflow?
Underflow?
Bit efficiency?
Representation(s) of 0?
Unique representations?
Rounding?

5.
What are some optimizations that can be made to the following function?

```
void cs33fun(char* Midterm, char* Grade, int* Final, int n) {

        for (int i = 0; i < (strlen(Midterm)); i++) {
                strcat(Grade, Midterm);

                for (int j = 0; j < n; j++)
                        for (int k = 0; k < i; k++)
                                Final[j] += strlen(Grade);
        }
}
```

There are many ways this function can be optimized, including but not limited to:

- The innermost loop can be replaced with the statement:

  ```
  Final[j] += i * strlen(Grade);
  ```

- Move `strlen(Midterm)` outside of the loop

There are several things students might be tempted to do based on an incomplete understanding of the lecture on Wednesday. However, they SHOULDN'T do the following:

- Based on "Procedure calls" - Move strcat out of the loop
  - Strcat is required for the logic of the function
- Based on "Procedure calls" - Move `strlen(Grade)` outside of the outermost loop (and nothing else)
  - The string `Grade` changes over each iteration of the outermost for loop
  - BUT can be moved outside of the middle loop
    - Since strlen(Grade) increments by strlen(Midterm) during each outermost iteration, can actually be moved outside the outermost loop if handled correctly