Problem 1: Floating Point
Write the minifloat representation of the decimal value \(-5/8\). Recall that minifloat is a (made-up) 8-bit floating point format with 1 sign bit, 4 exponent bits, and 3 mantissa bits. Write your answer as a bit pattern in these boxes:

Problem 2: Pointers and Generics
Recall our generic “swap” function from class (reproduced below). It is used to make two values trade places in memory, and is commonly used in sorting arrays. There’s a right way to call this swap function in normal circumstances, but we’re asking you to use it a bit “creatively” to achieve particular results. Note: what matters for the correctness of these results is that if you were to print the contents of what ptr1 and ptr2 point to (see comment in code), it would match the “after.”

```c
void swap_any(void *a, void *b, size_t sz) {
    char tmp[sz];
    memcpy(tmp, a, sz);
    memcpy(a, b, sz);
    memcpy(b, tmp, sz);
}
```

Complete the `mixup1` function to create this before & after result. Your solution must consist of ONLY completing the arguments of the one call to `swap_any`, as shown.

Before: After:
```
void mixup1(int *ptr1, int *ptr2) {
    swap_any(________________________,  
              __________________________,
              __________________________);

    //print what ptr1 and ptr1 point to
    //here, before returning
}
```
Complete the `mixup2` function to create this before & after result. Your solution must consist of ONLY completing the arguments of the one call to `swap_any`, as shown. In this case (part (c)), the third argument should not be edited other than to specify a single argument (that should be a standard type) to `sizeof()`.

Before:

<table>
<thead>
<tr>
<th>ptr1</th>
<th>ptr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

After:

<table>
<thead>
<tr>
<th>ptr1</th>
<th>ptr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

```c
void mixup2(int *ptr1, int *ptr2) {
    swap_any(____________________, __________________, sizeof(_______));
    //print what ptr1 and ptr1 point to here, before returning
}
```
Problem 3: Assembly and optimizations

Consider the following x86-64 code output by `gcc` using the settings we use for this class (-Og):

```assembly
<ham>:
    mov (%rdi),%eax
    lea (%rax,%rax,2),%esi
    add %esi,%esi
    mov $0x0,%ecx
    imul $0x31,%esi
    jmp L1
L3:
    lea (%rcx,%rax,1),%edx
    movslq %edx,%rdx
    mov %esi,(%rdi,%rdx,4)
    add $0x2,%eax
    jmp L2
L4:
    mov %ecx,%eax
L2:
    cmp $0x9,%eax
    jle L3
    add $0x3,%ecx
L1:
    cmp $0x9,%ecx
    jle L4
    mov $0xa,%eax
    retq
```

Refer back to this code to answer the questions in parts (a)-(d), on the following pages.
(a) (8pts) Fill in the C code below so that it is consistent with the above x86-64 code. Your C code should fit the blanks as shown, so do not try to squeeze in additional lines or otherwise circumvent this (this may mean slightly adjusting the syntax or style of your initial decoding guess to an equivalent version that fits). Your C code should not include any casting. Note that with the compiler set to –Og, some optimization has been performed. One thing you'll notice right away is that gcc chose not to create an actual eliza array, but instead kept track of its values in other ways. We will ask about optimizations in more detail in later parts of this question.

```c
int ham(int *burr) {
    int eliza[4];
    eliza[0] = 7;
    eliza[1] = 7;
    eliza[2] = 1;

    eliza[3] = ___________ * burr[0]; // part (b)

    for (int i = 0; i < ___________; i+=___________) {
        for (int j = ___________; j < ___________; j+=___________) {
            burr[___________] = eliza[0] * eliza[1] * eliza[2] * eliza[3]; // (c)
        }
    }

    if (eliza[0] > eliza[1]) { // part (d)
        return 8;
    }

    if (burr[0] < burr[1] && burr[0] > burr[1]) { // part (d)
        return 9;
    }

    return ___________;
}
```
(b) (2pts) Refer back to the C code, on the line marked for part (b). It reads:

```
eliza[3] = ... * burr[0];
```

**Name and explain** the instruction(s) that implement this product, and explain why gcc would choose to do it that way.

---

**Problem 4: Assembly and optimizations**

For parts (a) and (b), refer to the following x86-64 code output by gcc using the settings we use for this class (-Og):

```
<ham>:
40052d:       shl    $0x4,%edi
400530:       mov    %edi,%r9d
400533:       mov    %edi,%r10d
400536:       mov    $0x0,%eax
40053b:       lea    0x2(%rdi),%edi
40053e:       jmp    400562 <ham+0x35>
400540:       movslq %edx,%rcx
400543:       add    (%rsi,%r8,8),%rcx
400547:       movb   $0x58,(%rcx)
40054a:       add    %r9d,%eax
400550:       jmp    40055a <ham+0x2d>
400552:       mov    $0x0,%edx
400557:       movslq %r10d,%r8
40055a:       cmp    %edx,%edi
40055c:       jg     400540 <ham+0x13>
40055e:       sub    $0x1,%r10d
400562:       test   %r10d,%r10d
400565:       jg     400552 <ham+0x25>
400567:       repz retq
```
(a) Fill in the C code below so that it is consistent with the above x86-64 code. Your C code should fit the blanks as shown, so do not try to squeeze in additional lines or otherwise circumvent this (this may mean slightly adjusting the syntax or style of your initial decoding guess to an equivalent version that fits). Your C code should not include any casting. Note that with the compiler set to -Og, some optimization has been performed. We will ask about optimizations in more detail in part (b). There is an ASCII table as well as the 1-page x86 summary at the back of the exam. [FOR PRACTICE EXAM, REFER TO ASCII TABLE ONLINE IF NEEDED]

```c
int ham(int aaron, char **alex)
{
    int burr = _______________;

    for (int i = _______________ * ___________ /* see part (b) */;

        i > _______________; _______________)
        {

            for (int j = __________; j < _________________;

                _________________) {

                alex[i][j] = 'X';

                _________________ += _________________;
            }
        }
    return burr;
}
```

(b) Refer back to the C code for <ham>, on the line marked for part (b) (a multiply operator between two blanks). **Name and explain** the instruction(s) that calculate this multiplication, and how/why gcc optimized here:
For part (c), refer to the following C and x86 code:

```c
int eliza(char *peggy)
{
    int len = strlen(peggy);
    if (len == 8) return 8;
    else return len;
}
```

```
<eliza>: // optimized (-O2)
        sub $0x8,%rsp
        callq 400410 <strlen@plt>
        add $0x8,%rsp
        retq
```

(c) Refer back to the C code and assembly code for the function <eliza>. You’ll notice that although the C code includes an if statement, there are no conditional jumps in the assembly code. Explain how/why gcc optimized here:
Problem 5: Floats and bitwise ops

Beyoncé and Jay-Z set up a two bank accounts, one for each of their twins, and thought maybe minifloat would be a fittingly “mini” way to represent the account balances. Twin A’s account has a balance of $160.00, and Twin B’s account has a balance of $18.00 (don’t worry, they love them equally). Write each account balance in the spaces below.

Recall that minifloat is a made-up 8-bit floating point format with 1 sign bit, 4 exponent bits, and 3 mantissa bits, with a bias of 7.

(a) Write $160.00 in minifloat representation:

\[
t \text{twin}_a = \quad \]

Scratch work space:

(b) Write $18.00 in minifloat representation:

\[
t \text{twin}_b = \quad \]

Scratch work space:

(c) (3pts) After a few months, they decide that having two separate accounts is too cumbersome and they merge them into one account, containing the sum of the two individual accounts. Write the sum in minifloat representation:

\[
t \text{twin}_a + \text{twin}_b = \quad \]

Now write the sum \( t \text{win}_a + \text{twin}_b \) in decimal: __________________________

Scratch work space:
Problem 6: Assembly
Consider the following x86-64 code output by gcc using the settings we use for this class. This function calls another function, story, and you will be asked to reverse-engineer both of them.

For parts (a) and (b), fill in the C code below so that it is consistent with the x86-64 code above. Your C code should fit the blanks as shown, so do not try to squeeze in additional lines or otherwise circumvent this. This may mean adjusting the syntax, style, or expression of your initial decoding guess to an equivalent version that fits the structure of the provided C code. All int literals in your C code must be written in decimal.

(a) Fill in the schuyler function:

```c
int schuyler(int peggy)
{
    int angelica;
    int eliza = story(__________________________,
                      ____________________________,
                      ____________________________,
                      "helpless");

    ____________________________ *= 2;

    return ____________________________ + ____________________________;
}
```
(b) Now fill in the `story` function. Note that you aren't expected to have memorized the precise ASCII value of the letter 'f' that appears in the C code, but you should be able to infer its hexadecimal value in the x86-64 code, and thus be able to complete the line of code. All int literals in your C code must be written in decimal.

```c
int story(int raise, int *glass, char *freedom)
{
    if (____________________________ == 'f') {

        ______________________ = ______________________;
    } else {

        ______________________ = ______________________;
    }

    int tonight = ______________________;

    for (int i = ______________________; i >= 0; i -= ____________) {

        tonight += ____________;
    }

    return ______________________ * ______________________;
}
```