Assume we are running code on a 6-bit machine using two’s complement arithmetic for signed integers. A “short” integer is encoded using 3 bits. Fill in the empty boxes in the table below. The following definitions are used in the table:

\[
\text{short } sy = -3; \\
\text{int } y = sy; \\
\text{int } x = -17; \\
\text{unsigned } ux = x;
\]

Note: You need not fill in entries marked with “-”.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Decimal Representation</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>-6</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>-6</td>
<td>01 0010</td>
</tr>
<tr>
<td>ux</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tmax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-TMin</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

-TMin and Tmax are the minimum and maximum two’s complement numbers.
Consider the source code below, where M and N are constants declared with #define.

```c
int mat1[M][N];
int mat2[N][M];

int sum_element(int i, int j)
{
    return mat1[i][j] + mat2[i][j];
}
```

A. Suppose the above code generates the following assembly code:

```
sum_element:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 12(%ebp),%ecx
    sall $2,%ecx
    leal 0(%eax,8),%edx
    subl %eax,%edx
    leal (%eax,%eax,4),%eax
    movl mat2(%ecx,%eax,4),%eax
    addl mat1(%ecx,%edx,4),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

What are the values of M and N?
Consider the following C functions and assembly code:

```c
int fun1(int a, int b)
{
    if (a < b)
        return a;
    else
        return b;
}

int fun2(int a, int b)
{
    if (b < a)
        return b;
    else
        return a;
}
```

Which of the functions compiled into the assembly code shown?

---

A. On a x86 32-bit machine, Alice intends to use the expression \( \text{if } ((x \& \text{mask}) \neq 0) \) to test if the 5th bit of \( x \) from the right is one or not. (The rightmost bit of \( x \) is considered as the 0-th bit). The value of \( \text{mask} \) should be \[ \text{rectangle} \] (decimal).

B. Which of the following expressions generate the desired \( \text{mask} \) value in Question (A)? Select all that apply.

(a) \( 1 << 6 \)
(b) \( 1 << 5 \)
(c) \( \sim (1<<6) \)
(d) \( \sim (1<<5) \)
(e) \( 1 >> 26 \)
(f) \( 1 >> 27 \)

C. Please give the expression which sets the 5th bit of \( x \) to be one and leave the rest of the bits of \( x \) unchanged. Your expression should only use the \( \text{mask} \) value in Question (A) and no other constants.
Consider the following assembly code for a C for loop:

```assembly
loop:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%ecx
    movl 12(%ebp),%edx
    xorl %eax,%eax
    cmpl %edx,%ecx
    jle .L4

.L6:
    decl %ecx
    incl %edx
    incl %eax
    cmpl %edx,%ecx
    jg .L6

.L4:
    incl %eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Based on the assembly code above, fill in the blanks below in its corresponding C source code. (Note: you may only use the symbolic variables x, y, and result in your expressions below — do not use register names.)

```c
int loop(int x, int y)
{
    int result;

    for (____________; __________; result++) {
        __________;
        __________;
    }

    __________;
    return result;
}
```
void foo(int x)
{
    printf("address of x is \%p\n", &x);
    bar(x-1);
    return;
}

void bar(int y)
{
    printf("address of y is \%p\n", &y);
    return;
}

void foo2()
{
    bar2();
}

void bar2()
{
    char buf[10];
    gets(buf);
}

A. Suppose we invoke the function foo many times. What is the relationship of the address of x and that of y in resulting printf statements?
(a). address of x is always less than address of y.
(b). address of x is always greater than address of y.
(c). address of x is sometimes less than address of y and sometimes greater than address of y.

B. The gets(s) function reads a line from stdin into the buffer pointed to by s and does not check for buffer overrun. Suppose we invoke the function foo2 and the user types in some line longer than 10 characters with the intent of exploiting the buffer overrun to execute malicious code by overwriting a return address. Which function would that return address have been pointing to if the attack had not occurred? (That is, which function would we have returned to).