1.) **Rewrite the following C function in x86-64 assembly. Make sure to follow the register and procedure call conventions.**

```c
long fcn(long *ptr, long count) {
    if(count <= 0){
        return 0;
    }
    else{
        return ptr[0] + fcn(ptr+1, count-1);
    }
}
```

**Notes:** *using the standard C calling conventions, the first function argument is always stored in register $rdi, the second function argument in register $rsi, and the return value in register $rax.*

```assembly
fcn:
    movl $0, %eax
    cmpq $0, %rsi
    jle done
    movq (%rdi), %rdx
    pushq %rdx
    addq $8, %rdi
    subq $1, %rsi
    callq fcn
    popq %rdx
    addq %rdx, %rax

done:
    retq
```
2.) Consider the following C source code where a two dimensional array of longs has been declared in the following manner.

```
long array[M][N];
```

This declaration of array is followed by its initialization (not shown). Then the C code (left) summing elements in column j is executed. The x86-64 assembly code (right) is a translation of that C code.

```c
long result = 0;
long j = 2;
for(long i = 0; i < M; i+=1){
    result += array[i][j];
}
```

```
movq $0, $r8     # result
movq $0, $rdx   # i
jmp cnd
loop:
    movq $2, $rax  # j
    movq $rdx, $rsi
    shl $0x3, $rsi
    addq $rsi, %rax
    addq array(, $rax, 8), %r8
    addq $1, $rdx
    cnd:
        cmpq $0x4, $rdx
        jle loop
```

What are the values of M and N?

On each loop iteration, we multiple i by 8. That means there must be 8 elements in each row.

The combination of the cmpq and jle indicates that the last time we jump back to the loop label is when i == 4. That means when i is 5, we break out of the loop. So, i < 5 is the C code condition.

\[ M = 5, N = 8 \]
3.) Consider the following C code.

```c
struct item{
    char c;
    int num[2];
    long val;
    int id;
};
struct item singleItem;
struct item array[10];
```

For the questions below, assume all alignment and size requirements for structs discussed in the textbook are followed.

a.) Assume the address of the `singleItem` struct instance has been loaded into register `%rdi`. Complete the following x86-64 instruction that stores the value 4 into the memory location for `num[1]`.

```
movl $4, ______________
```

Offsets:
- `c` 0
- `num` 4
- `val` 16
- `id` 24

```
movl $4, 8(%rdi)
```

b.) Assume the starting address of `array` has been loaded into register `%rsi`. Complete the following x86-64 instruction that stores the value 10 into the memory location for `array[2].val`.

```
movq $10, ______________
```

Taking into account the alignment for each of the fields, `id` is in bytes 24-27. Thus, the size is 28 bytes. However, the total size of the struct has to be a multiple of the largest type, which is 8. That means the struct has to be 32 bytes.

The element at `array[2]` will start at 2*32 bytes after the beginning of the array. The `val` field is a 16 bytes from the start of the `array[2]` instance. So, 2*32+16 = 64+16 = 80.

```
movq $10, 80(%rsi)
```