1. Description of each MysteryFunction

MysteryFunction1: Calculates the power of a number - using two parameters, a base and an exponent. Then, the function will return the base raised to the exponent.

e.g. MysterFunction1(5,3) = 125 //5^3 = 125

MysteryFunction2: Isolates and checks individual bits of an integer to perform bitwise manipulation. It shifts through each bit and combines them in a specific way, ultimately returning the modified value. This can be used to manipulate the bits of the integer in different ways, such as creating a mask or flipping certain bits.

e.g. MysteryFunction2(5) = 2684354560

MysteryFunction3: Finds the maximum value in an array of integers

e.g. MysteryFunction3 (max in array) = 7 //array: {1,3,7,05}

MysteryFunction4: Counts the number of 1s in the binary form of an unsigned long integer, using bitwise operations to isolate and count the bits until all bits have been processed.

e.g MysteryFunction4(15) = 4; //binary # 1111 has four 1s

MysteryFunction5: Computes the number of differing bits between two unsigned integers, using the XOR operation to find bits that differ and counts them until all bits have been processed.

e.g. MysteryFunction5(15, 0) = 4 //binary # 1111 differs from 0000 by 4 bits

2. Discussion on the approach I used to figure out what the MysteryFunction was

I carefully studied the assembly code to understand the purpose of **MysteryFunction1**. The function initializes the result to 1 and then multiplies it by the base while decreasing the exponent, indicating it calculates the power of a number. Testing the function with various outputs confirmed the correct behavior, and the relationship between the exponent and the number of multiplications performed became clear.

When investigating **MysteryFunction2**, I focused primarily on analyzing the bit manipulation it performs. The function uses **shifts**, along with **AND** and **OR** operations to process the bits of the input integer. For each bit set to 1, it reverses its position, placing a 1 in the matching spot on the opposite end of a 32-bit int. This effectively mirrors the positions of the set bits. This function was the hardest to figure out because its use of bit shifting initially threw me off. Although I had learned about bit manipulation before, I hadn't immediately thought of applying it in this context, which made understanding the function more challenging compared to the others. However, after running several test cases, I observed how different integers resulted in varying output values, helping me conclude that the function transforms the input based on its bit patterns.

For **MysteryFunction3**, I concentrated on the logic for processing arrays. The function checks each element against a stored maximum, iterating through the array to find the largest value. Its purpose is to find the maximum value in an array, which I confirmed through testing with various arrays and verifying the results.

When analyzing **MysteryFunction4**, I discovered it counts the number of set bits in the input number. The repeated **AND** and **shift** operations suggest a systematic approach to isolate each bit, allowing for a count of how many set to 1. I tested this function with different integers and can confirm that it returns the correct number of set bits after processing all bits.

Finally, **MysteryFunction5** directed my attention to focusing on the differing bits between two integers. By examining how it utilizes the XOR function, I recognized that this function identifies which bits differ between two numbers. Testing with various pairs of integers provided clear evidence of how many bits differ, confirming the function's behavior.