Question 6: Derive the O() for the Recursive vs. Non-Recursive Fibonacci Function. See repository.

Given: Function fibAray() with an argument of integer N, and the function fibLoop() with an argument of integer N.

fibAray(int n) Approach: There are two main parts to consider when taking the BigO of the fibAray function. The first being the array "int array[n+1]" on line 66. This array is created with size n + 1, which means that the time it takes to allocate the memory for the size is directly related to the size n, making this operation timing O(n).

The second important part to consider with regards to BigO is the functions for-loop. This loop, "for(int i=2;i<=n;i++)", performs constant time operations related to the size of the input n. The inner operations of the loop, "array[i]=array[i-1]+array[i-2]", are also a constant number of operations related to argument n. Both aspects run at a constant time related to size n, meaning that the operation timing is O(n)

fibRec(int n) Approach: The recursive countertype of the Fibonacci algorithm differs drastically despite being a shorter function (see repository). The drastic increase in computation time comes from "fibRec(n-1)+fibRec(n-2)", which creates two recursive calls to fibRec of inputs n until a base case is met. For a better example, the next recursion to find fibRec(n-1) calls fibRec() twice, for fibRec(n-2) and fibRec(n-3). This can be visualized as something of a tree pattern, creating two nodes for each call n. The greatest takeaway being for each n, there are two 2 recursive calls being made. That leads to the operation timing being $O\left(2^n\right)$.

Therefore The conclusion can be made that the function fibAray() with argument n computes at O (n) timing, and the function fibRec() with argument n computes at $O\left(2^n\right)$.