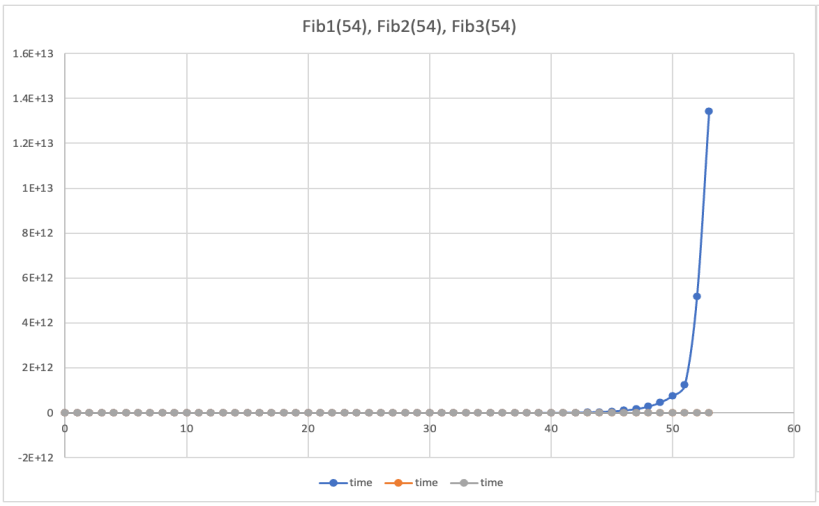

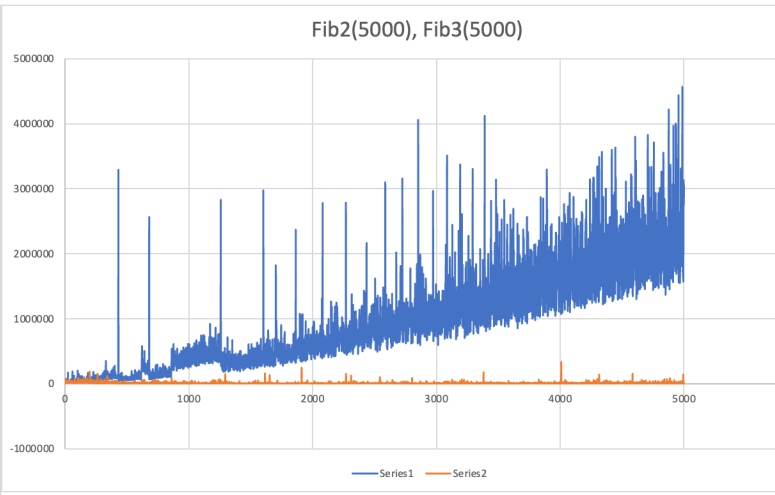

For the first graph, we can clearly see that an execution of Fib1() results in a much higher running time than Fib2() and Fib3() since it takes more nanoseconds to compute all 54 values.
In our opinion, the difference in the running time of the algorithms is due to the steps required for each algorithm to be executed. In the first algorithm, the fibonacci number is calculated by recursively computing the previous two numbers (fib(n-1)+fib1(n-2)). Essentially, the larger the value of n, the more steps the algorithm requires since it recursively calls itself with smaller values until n < 2.

Fib 2 is faster than Fib1 because there are no such recursive calls that we mentioned about Fib1, it simply uses a for loop and so its run time is O(n)--since it is dependent on the value of n– which is faster than fib1 which has a run time of O(2^n).

The graph for Fib3 shows that Fib3 has a faster run time than fib1 and fib2, we believe this is due to the fact that fib3 utilizes an array and so it doesn't have to keep recalculating older values and remembering them as they are simply stored in an array and referenced as the program executes. This leaves fib3 with a runtime of O(1) which is faster than fib1 and fib2.

For **fib1**(54), the big-oh is most likely O(2^n) since the curve is shown to grow exponentially and the number of nanoseconds increases for values of n, meaning as n increases, the algorithm takes longer to compute.
For **fib2**(54), the big-oh is most likely O(n), since the time taken for each value of n is shown to grow linearly depending on the value of n. We can see in the graph for **fib2(5000)** that since n is a much larger number being 5000 the growth reflects this.
For **fib3**(54), the big-oh is most likely O(1) making it the fastest algorithm and it runs in constant time. Just looking at the curve we can see that Fib3 in both the 54 input version and **fib3(5000)** input version is very flat which is in line with the graph of O(1). What further proves that Fib3 is O(1) is that even when the n input is changed to 5000 the time it takes to calculate each value even at the very large n's is around the same. Meaning Fib3 runs in constant time O(1).
From the three different algorithms, I prefer using fib3. Fib3 is the fastest out of the different algorithms, it also has the same runtime for each fibonacci number calculation regardless of input size. So specifically if the input size was very large I would use Fib3.

Part 2:

1) **Loop Invariant:** j = fib2(n) and i = fib2(n - 1)

2) **Proof of correctness:**

Loop Invariant: j = fib2(n) and i = fib2(n - 1)

Base Case:
Initially, when n = 0, j = fib2(0) and i = fib2(n -1). The loop is not executed since k is initially 1 and satisfies the condition in the for loop since k > n at the start of the loop so the algorithm returns j which is 0.

When n = 1, j = fib2(1) and i = fib2(0). The loop executes once and adds i to j and stores that value in j, returning the value of j which is 1. Thus this loop invariant holds for the cases n = 0, and n = 1.

Inductive hypothesis: Assume the loop invariant holds for all non negative values k such that j = fib2(k), i = fib2(k-1) and k <= n.

Inductive Claim: The loop invariant holds for k + 1 values such that j = Fib2(k + 1) and i = Fib2(k).

Proof:
Initially, before the loop is reached for the first time, i is set to the value 1 and j is set to be the value 0.

Assuming that the loop invariant is correct before an execution of the k + 1 iteration, we can infer from the inductive hypothesis that the value j is defined as j = fib2(k) and the value i is defined to be fib2(k-1).

For the k + 1th iteration, i and j are updated to be:

j = i + j = fib2(k) + fib2(k + 1) = fib2(k + 2)
i = j - i = fib2(k + 2) - fib2(k) = fib2(k + 1)

So n = j + i where j = fib2(k+2) and i = fib2(k+1) and so our loop invariant holds since i is 1 less than j.

Thus our loop invariant satisfies our claim and holds for k + 1 values

From the condition in the loop, we see that the invariant holds for all non negative values of n and the loop runs as long as k <= n. Once k > n, where n is our input value, the loop terminates and outputs j as the fib2(k) value and i as the fib2(k-1) value as required.