Java Fibonacci Methods Documentation

I. Implemented methods

In this assignment, I implemented methods to calculate the n^{th} Fibonacci number. The idea is that, let fib(n) defines the function, then fib(0) returns 0, fib(1) returns 1, and so on. This solution is created based on two assumptions: (1) the Fibonacci sequence starts from 0, 1, 1, 2, 3, 5, and so on; (2) we are counting zero-based index, which means the first number in the sequence is the 0^{th} Fibonacci number. The two methods are implemented as following:

1. The recursive approach:

   With all recursive functions, we define two things: (1) base case, and (2) sub problems.

   - Base case: when n <= 2 → return n
   - Sub problems: fib(n) = fib(n – 1) + fib(n – 2), so the problem of calculating the n^{th} Fibonacci is broken down into two sub problems: calculating the $(n - 1)^{th}$ and $(n - 2)^{th}$ Fibonacci numbers.
   - Pseudo code:

   ```
   fib(n) {
       if (n == 0) return 0;
       if (n == 1) return 1;
       if (n == 2) return 2;
       return fib(n – 1) + fib(n – 2);
   }
   ```

   - Runtime complexity: **$O(2^n)$**. Each time we go down one level of the recursion tree, we split into two branches, and we do so for n levels.

2. The loop approach:

   This approach is based on the dynamic programming technique; that is, the solution at position n is calculated using the pre-calculated solution at position n – 1

   - Idea: use an array of n + 1 elements to store Fibonacci numbers up to n (because we are doing zero-based). We hard coded the first 2 elements of the array as 0, 1. From the 3^{rd} element, every element can be calculated using the sum of the previous two elements.

- Pseudo code:

```
int[] fibos = new int[n + 1];
fibos[0] = 0; fibos[1] = 1;
for i = 2 → n + 1:
    fibos[i] = fibos[i – 1] + fibos[i – 2]
return fibos[n]
```

- Runtime complexity is **O(n)**. Each calculation can be done in O(1), and we do so n times.

II. Runtime comparison

From my experiment, when the first program runs on my computer, the program almost cannot run as n approaches 40. Therefore, for the experiment to compare the run time of the two algorithms, I will use 35 as a threshold for n.

Detailed steps:

- I run both algorithms with n ranging from 1 to 35, with step of 2 (that is, n = 1, 3, 5, 7, …, 35).
- I record the run time (in nanoseconds) of them with each input n.
- I use Java I/O libraries to write the output to a CSV file. A table version of the csv file is as following:

n	Recursive Runtime	Loop Runtime
2	2233	1288
4	908	864
6	1186	577
8	2998	551
10	9312	673
12	23731	706
14	55587	2900
16	707754	2599
18	708658	2269
20	1303890	1942
22	3019082	3047
24	248457	1860
26	597166	2615
28	85662105	8932
30	4428111	8222
32	11016648	12590
34	87053948	10254

- With the help of Microsoft Excel, I created a line chart to compare the run time of the two algorithms. A copy of the chart is as following: