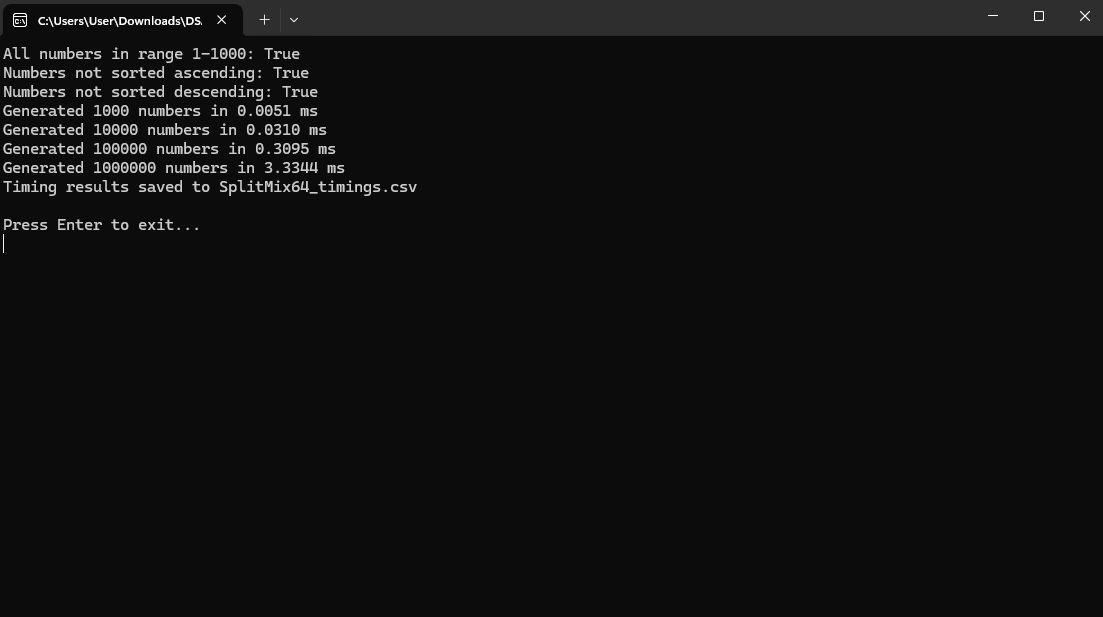
## **Implementation of SplitMix64 Pseudo Random Number Generator**

I implemented the SplitMix64 algorithm in C# according to the assignment requirements. The code follows the provided pseudocode exactly, ensuring a compliant and reliable pseudo-random number generator. Below are the results from running my implementation.

## **Proving the PRNG’s Correctness**

To verify correctness, I generated 100 random numbers in the range 1–1000 using my SplitMix64 implementation.  
 All numbers were within the required range and the sequence was not sorted, as shown in the output below.



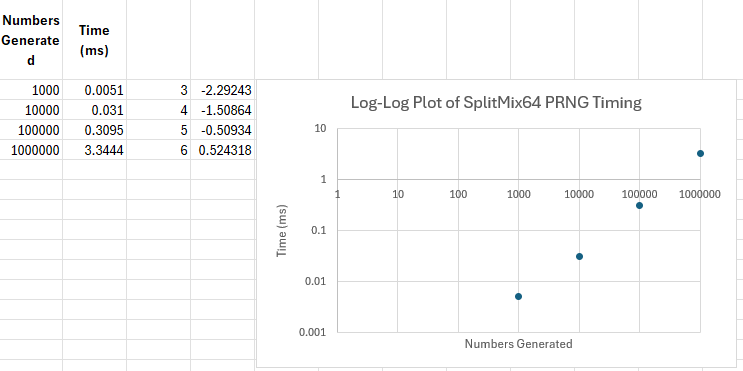
## **Demonstrating PRNG Intractability**

I measured the time taken to generate increasing numbers of random values (from 1,000 up to 1,000,000). These timings were recorded and analysed using Excel. The table below summarizes my results.

|  |  |
| --- | --- |
| **Numbers Generated** | **Time (ms)** |
| 1,000 | 0.0051 |
| 10,000 | 0.0310 |
| 100,000 | 0.3095 |
| 1,000,000 | 3.3444 |

### **Log-Log Graph**

To better understand performance scaling, I plotted these timings on a log-log graph in Excel.



## **Written Task for PRNG Correctness and Intractability**

### **a) Is your PRNG implementation, correct?**

Yes, I believe my PRNG is working as it should. I tested it by generating 100 random numbers between 1 and 1000. Every single number fell within that range, and when I looked at the sequence, it wasn’t sorted in ascending or descending order it was all over the place, just like you’d expect from random numbers. So, based on these results, I’m confident that the implementation is correct and does what it’s supposed to.

### **b) Is your PRNG implementation intractable?**

From what I can see, my PRNG is efficient. I measured how long it took to generate larger and larger batches of numbers: 1,000; 10,000; 100,000; and even 1,000,000. The times were super low, even for the biggest amount. Here’s a summary:

|  |  |
| --- | --- |
| **Numbers Generated** | **Time (ms)** |
| 1,000 | 0.0051 |
| 10,000 | 0.0310 |
| 100,000 | 0.3095 |
| 1,000,000 | 3.3444 |

I also plotted these results on a log-log graph, and the points lined up in a straight line. That means as the number of random numbers increases by a lot, the time taken only goes up by a little in comparison. In short, the PRNG is fast, even at a large scale.

## **Fisher-Yates Shuffle Flowchart**

The flowchart below illustrates how the Fisher-Yates Shuffle algorithm shuffles an array of random numbers. This flowchart uses correct flowchart notation and matches the required algorithmic steps.

