An Analysis of Java Loop Execution Time vs Loop Length

Christiffer Vogts

In the context of computational randomness and performance benchmarking, this study investigates the relationship between loop length and execution time in Java. The primary objective was to explore whether a simple randomized output mechanism could be implemented in Java using a for loop, and to determine how the length of such a loop influences the time required for execution. This inquiry emerged from a broader goal: to develop a method for generating outputs that exhibit characteristics of "true randomness" within the constraints of Java's pseudo-random number generation.

A basic Java program was constructed to execute a loop of variable length, where each loop calculates a random number and outputs the result to the console. The execution time for each loop length was recorded using Microsoft Excel to identify trends and correlations. Preliminary observations indicated that the time required to complete the loop increased disproportionately with the number of iterations, suggesting a non-linear growth pattern.

The experimental setup involved a single Java class, as shown in Figure 1, executed on a Dell XPS 9530 laptop. The code is based off a loop of user defined length and performs a compound arithmetic operation involving three calls to Math.random(). Specifically, the expression:

((Math.random() * 100 + 1) * (Math.random() * 100 + 1)) / (Math.random() * 100 + 1)

was chosen arbitrarily to simulate a computationally non-trivial randomization process. Each iteration calculates this expression, assigns the result to a variable, outputs it to the console, and then increments the variable once more. The total execution time of the loop is measured using System.currentTimeMillis() before and after the loop execution.

```
package main;
public class Main {

	public static void main(String[] args) {


		double numb_on = 0;
		int length = 1000;
		long loop = System.currentTimeMillis();
		for (int i = 0; i < length; i++) {
			numb_on =((Math.random() * 100 + 1) * (Math.random() * 100 + 1)) /
(Math.random() * 100 + 1);
			System.out.println(numb_on);
			numb_on++;
			}
		System.out.println(System.currentTimeMillis()-loop);
	}
}
```

Figure 1. Java code used to measure loop execution time as a function of loop length.

Following each execution, the recorded time was manually entered into an Excel spreadsheet alongside the corresponding loop length. This enabled the construction of a time-series dataset for further analysis. Table 1 presents the raw data collected across multiple trials, while Graph 1 visualizes the relationship between loop length and execution time.

Upon plotting the average execution time against loop length, a clear exponential trend emerged. Graph 2 illustrates this trend, with the best-fit exponential curve described by the equation:

$$Y = 835.71\ e^{-1.248x}$$

Where y = average time and x = loop length

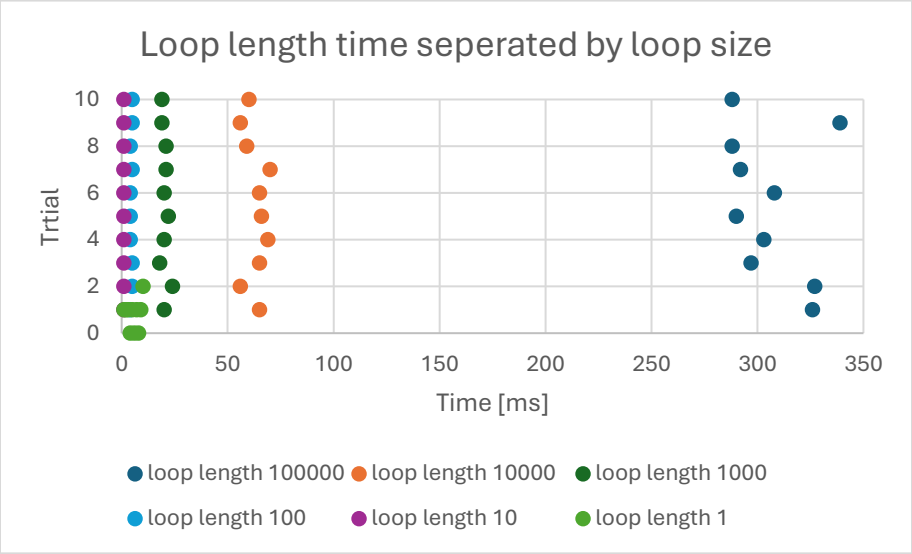
This suggests that the time complexity of the loop, under the specific conditions of randomized arithmetic and console output, grows exponentially with respect to the number of iterations. It is important to note that this behavior is influenced not only by the arithmetic operations but also by the overhead associated with console I/O, which is known to be relatively expensive in Java.

Java runtime for loop length						
Tria\loop length	100000	10000	1000	100	10	1
1	326	65	20	4	1	1
2	327	56	24	5	1	1
3	297	65	18	5	1	1
4	303	69	20	4	1	0
5	290	66	22	4	1	1
6	308	65	20	4	1	0
7	292	70	21	5	1	1
8	288	59	21	4	1	0
9	339	56	19	5	1	1
10	288	60	19	5	1	2
avg	305.8	63.1	20.4	4.5	1	0.8
STD	18.62973	5.043147	1.712698	0.527046	0	0.632456

Table 1. the table of all the data values where time is measured in milliseconds

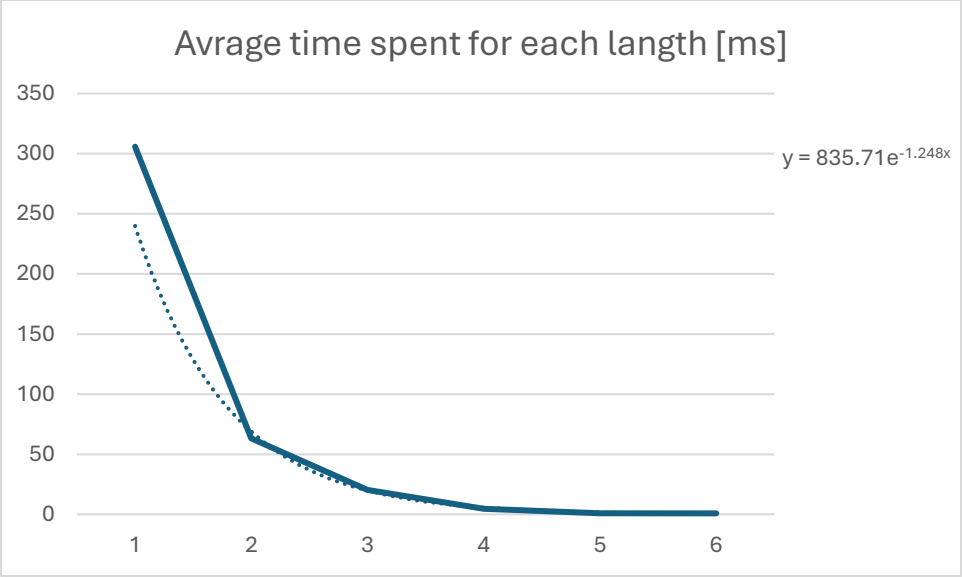
The findings raise several important considerations for developers and researchers working with randomized algorithms or performance sensitive applications in Java. First, while the use of Math.random() provides sufficient pseudo-randomness for many applications, the

computational cost of repeated random number generation and output operations must be accounted for, especially in high-frequency or real-time systems.



Graph 1. A visualization of the time each trial took to complete.

Second, the exponential growth in execution time suggests that loop-based randomization strategies may not scale efficiently. This has implications for algorithm design, particularly in contexts where large datasets or high iteration counts are involved. Future work could explore alternative randomization techniques, such as buffered output or parallel processing, to mitigate performance bottlenecks.



Graph 2. A visualization of the mean speed with the trend line shown to the left.

This experiment demonstrates that the execution time of a Java for loop performing randomized arithmetic and console output increases exponentially with loop length. While the initial goal was to explore randomness, the study revealed critical insights into the performance characteristics of loop-based operations in Java. These findings underscore the importance of considering computational overhead when designing randomized systems and provide a foundation for further exploration into efficient randomization and benchmarking techniques.