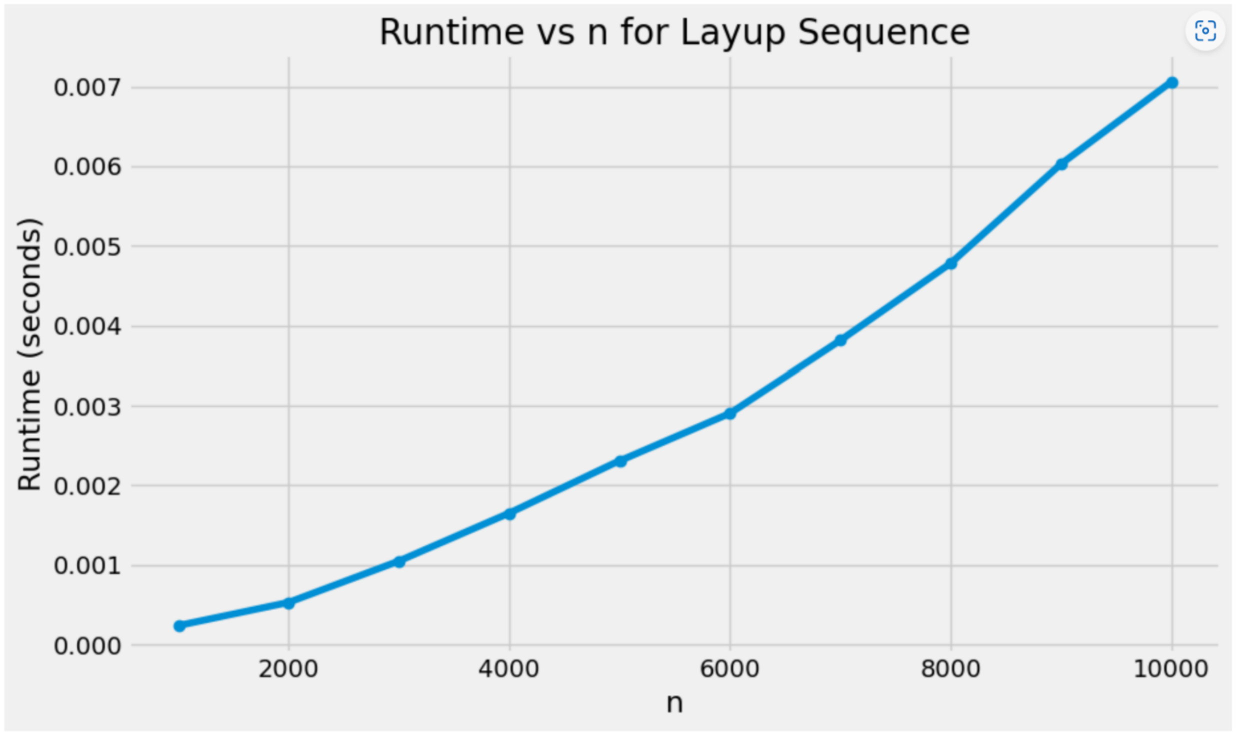
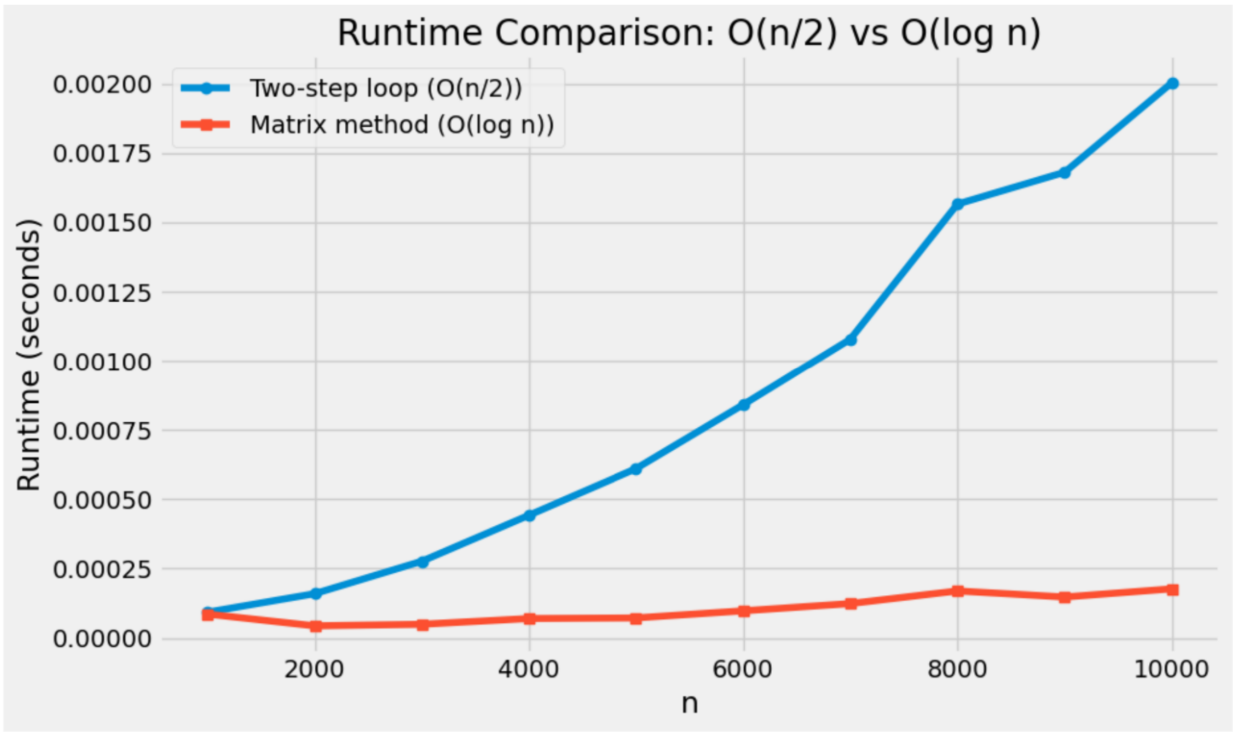
I first approached the problem using recursion because the pattern reminded me of the Fibonacci sequence. But as soon as I tried it with larger values of n, I ran into stack overflow issues. So I switched to a loop-based solution that just kept track of the last two values in the sequence. This worked much better and avoided crashing, but it still took some time to run for larger inputs since it processed one step at a time.

To improve that, I realized I could actually calculate two values in each loop, one for the next odd index and one for the next even index, since each value depends on the two before it. This cut the number of loop steps in half, and made the program run significantly faster while still keeping the code simple and efficient.



I understand the expectation was to spend around an hour on this task, but since there was no strict time limit, I took the opportunity to explore the problem a bit deeper. I ended up going beyond that time to find a more optimized solution. I implemented it in Python because its built-in support for matrix operations allowed me to simplify some of the logic. This helped me focus more on the structure of the solution and less on the boilerplate code. While it took longer, I learned a lot in the process and was able to come up with a cleaner and more efficient approach.



The plot above was made in Python to compare how long each version takes to run. You can clearly see that the matrix approach is much faster, especially as the value of n gets bigger. While the loop version gets slower the larger n is, the matrix method stays quick and consistent. This makes the matrix version a better option when performance matters or when working with really large numbers.