Activity 1. currentTimeMillis()

* Calculate how many more years we can continue using this way of counting. Explain what you did to calculate it.

This built-in Java method returns the time in milliseconds since January 1, 1970, 0:00 UTC as a 64-bit long, which has a maximum value of 9,223,372,036,854,775,807. This means we could keep using this system for another

(9,223,372,036,854,775,807 / 1,000 / 60 / 60 / 24 / 365) – 55 = **292,471,153.7**

years before it overflows.

That sounds pretty good!

Activity 2. Vector2.java

* Why does the measured time sometimes come out as 0?

It comes out as 0 because the algorithm has indeed taken less than a millisecond to run, at least when n is less than 100,000.

* From what size of problem (n) do we start to get reliable times?

We start to get reliable times (i.e. times over 50 ms), at approx. n = 13,000,000.

Activity 3. Vector4.java and beyond

* What happens with time if the problem size is multiplied by 2?

The execution time doubles with each step.

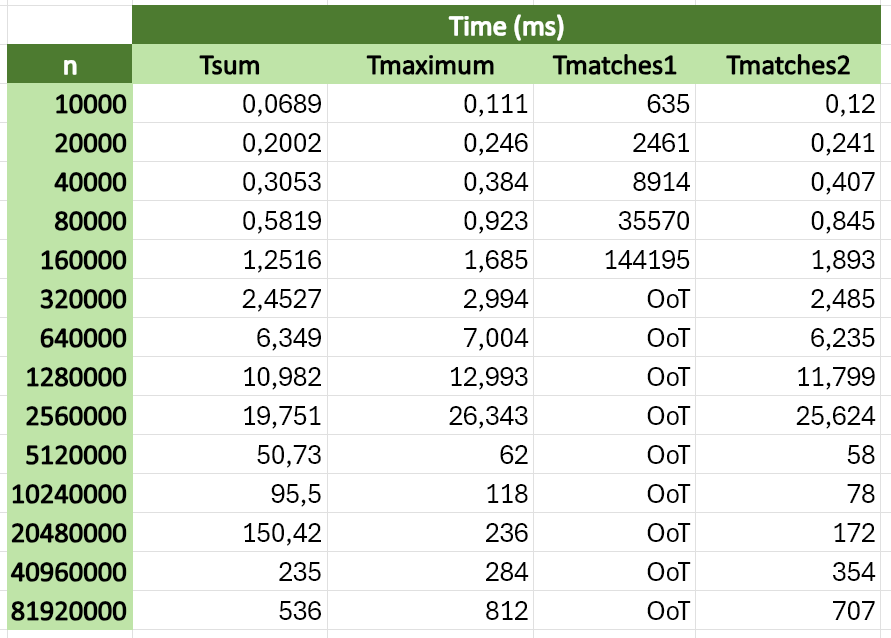
* What happens with time if the problem size is multiplied by a value k other than 2? (try it, for example, for k=3 and k=4 and check the times obtained)

The time is also multiplied by k each time the problem size increases.

* Explain whether the times obtained are those expected from the linear complexity O(n)

Yes, they are. As the size increases, the time grows along with it linearly.

**----- TABLE -----**



* Indicate the main features (processor and memory) of the computer where times have been measured.

My computer has an AMD Ryzen 9 6900HX 3.3Ghz processor and 32 GB of RAM.

* Once both tables are filled in, conclude whether the times obtained meet what was expected, given the computational time complexity of the different operations.

They all seem to follow a linear growth pattern, which is what the algorithms would suggest, except for **matches1**, which should be quadratic. I’m not sure why that is...