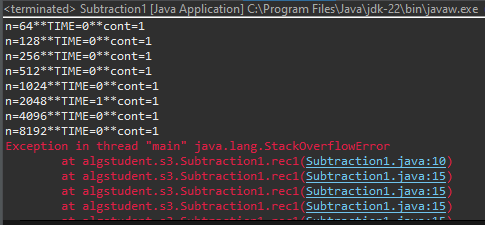
Activity 1. D&C by Subtraction

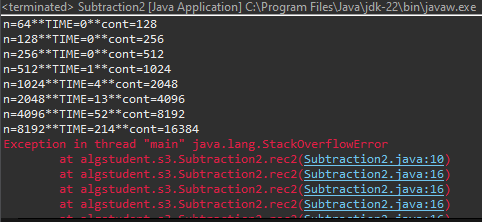
* After analyzing the complexity of the three previous classes, you are not asked to make the timetables, but to reason whether the times match the theoretical time complexity of each algorithm.

The algorithms in Subtraction1.java and Subtraction2.java are respectively linear and quadratic in their time complexity, although the empirical data is so small in the first one as to make the trend unnoticeable, and Subtraction2.java looks linear in the first few iterations; we’d have to measure them with a lot greater problem sizes for their real growth trends to become apparent.

* For what value of n do the Subtraction1 and Subtraction2 classes stop giving times (we abort the algorithm because it exceeds 1 minute)? Why does that happen?

They both seem to overflow once n > 8192, which is way before the times get even close to a minute. This happens because of the massive amount of times the method calls itself, therefore pushing more information to the stack than it can actually hold.





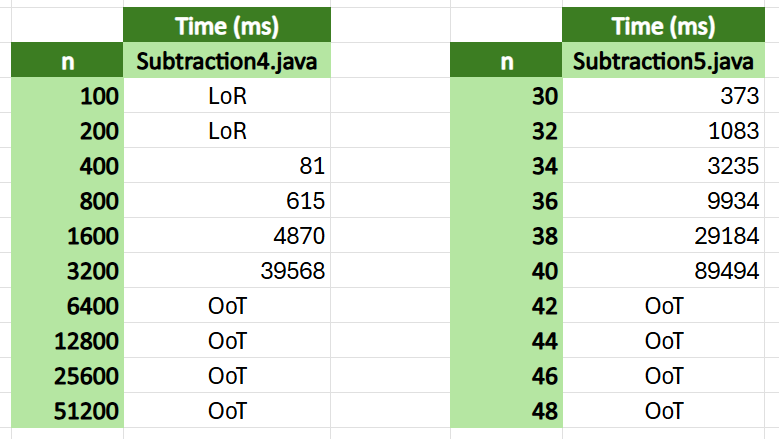
* How many years would it take to complete the Subtraction3 execution for n=80? Reason the answer.

Subtraction3.java, however, is exponential (O(2)). Taking [n = 30, t = 38,051 ms] as a reference point, we get that:

Which is about **1,358.5 million years.** Efficient!

* Implement a Subtraction4.java class with a complexity O(n3 ) and then fill in a table showing the time (in milliseconds) for n=100, 200, 400, 800, ... (until OoT).
* Implement a Subtraction5.java class with a complexity O(3n/2) and then fill in a table showing the time (in milliseconds) for n=30, 32, 34, 36, … (until OoT).

The times for Subtraction4.java and Subtraction5.java are as follows:



* How many years would it take to complete the Subtraction5 execution for n=80? Reason the answer.

For n=80, Subtraction5.java would take

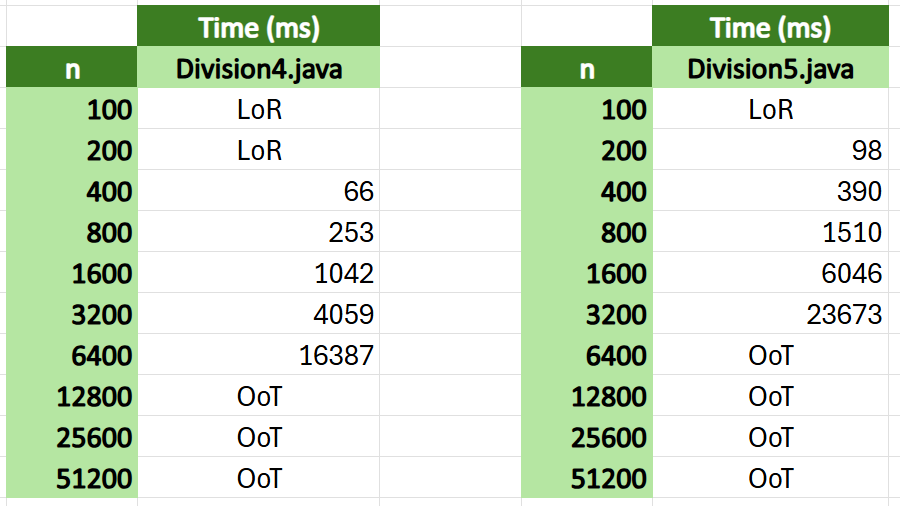
Or approximately **9,894.9 years.** Hey, at least it’s better than Subtraction3.java...

Activity 2. D&C by Division

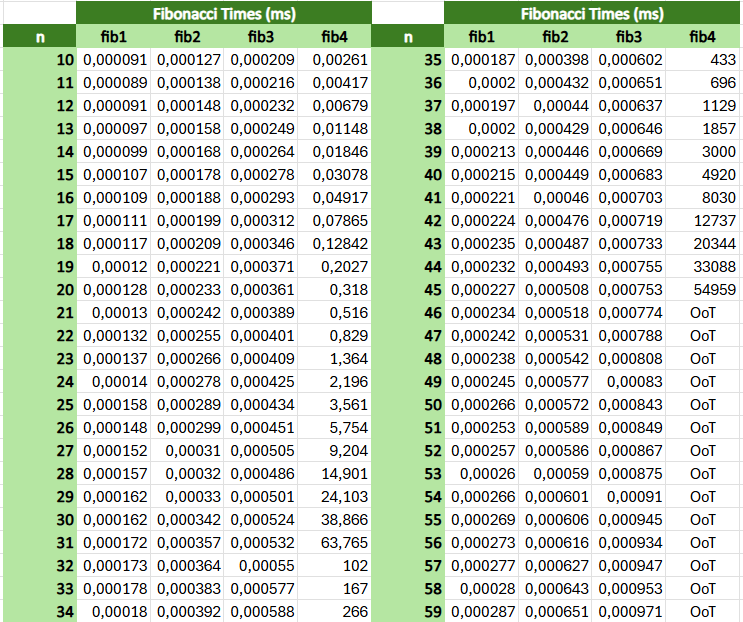
* After analyzing the complexity of the three previous classes, you are not asked to make the timetables, but to reason whether the times match the theoretical time complexity of each algorithm.

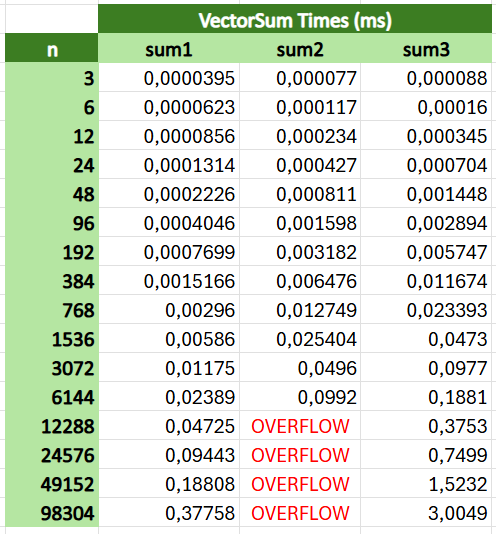
Indeed, they do. Division1.java and Division3.java are linear, as the times would suggest. Division2.java, on the other hand, is O(n\*log n), which is also coherent with the data when measured.

* Implement a Division4.java class with a complexity O(n^2 ) (with a<b^k ) and then fill in a table showing the time (in milliseconds) for n=1000, 2000, 4000, 8000, … (up to OoT).
* Implement a Division5.java class with a complexity O(n^2 ) (with a>b^k ) and then fill in a table showing the time (in milliseconds) for n=1000, 2000, 4000, 8000, … (up to OoT).



Activity 3. Two basic examples





* After analyzing the complexity of the various algorithms within the two classes, executing them and afterwards putting the times obtained in a table, compare the efficiency of each algorithm.

For Fibonacci, the algorithms are of decreasing efficiency: the first three are O(n), whereas the last one is O(1.6). The VectorSum algorithms, on the other hand, are all linear, with a clear preference for the first one. The second one might be better than the third one if it didn’t overflow, but alas, I guess we’ll never know.

Activity 4. Pétanque championship