Session 1.1

UO283069

Measuring execution times

1. How many more years can we continue using this way of counting?

Since the functions returns a long it can only return numbers on the range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

And since until this moment the current number of milliseconds that have passed between January 1, 1970, and nowadays is 1644772522798 milliseconds, we still have 9.223.370.392.082.247.202 milliseconds left that can be represented (292 million years).

2. What does it mean that the time measured is 0?

That the actual value is almost 0, or that its execution time is very low and takes less than a millisecond.

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

From higher sizes, the higher the size of the n the more reliable the time is going to be, although there is a limit, since very high numbers for n might end up taking a lot of time, the adequate size for a problem of complexity n is around 2430 and 196830, since (even if they are high numbers) a problem of complexity O(n) should not take too much time to solve then (around 10 seconds at maximum and 1 millisecond minimum)

Grow of the problem size

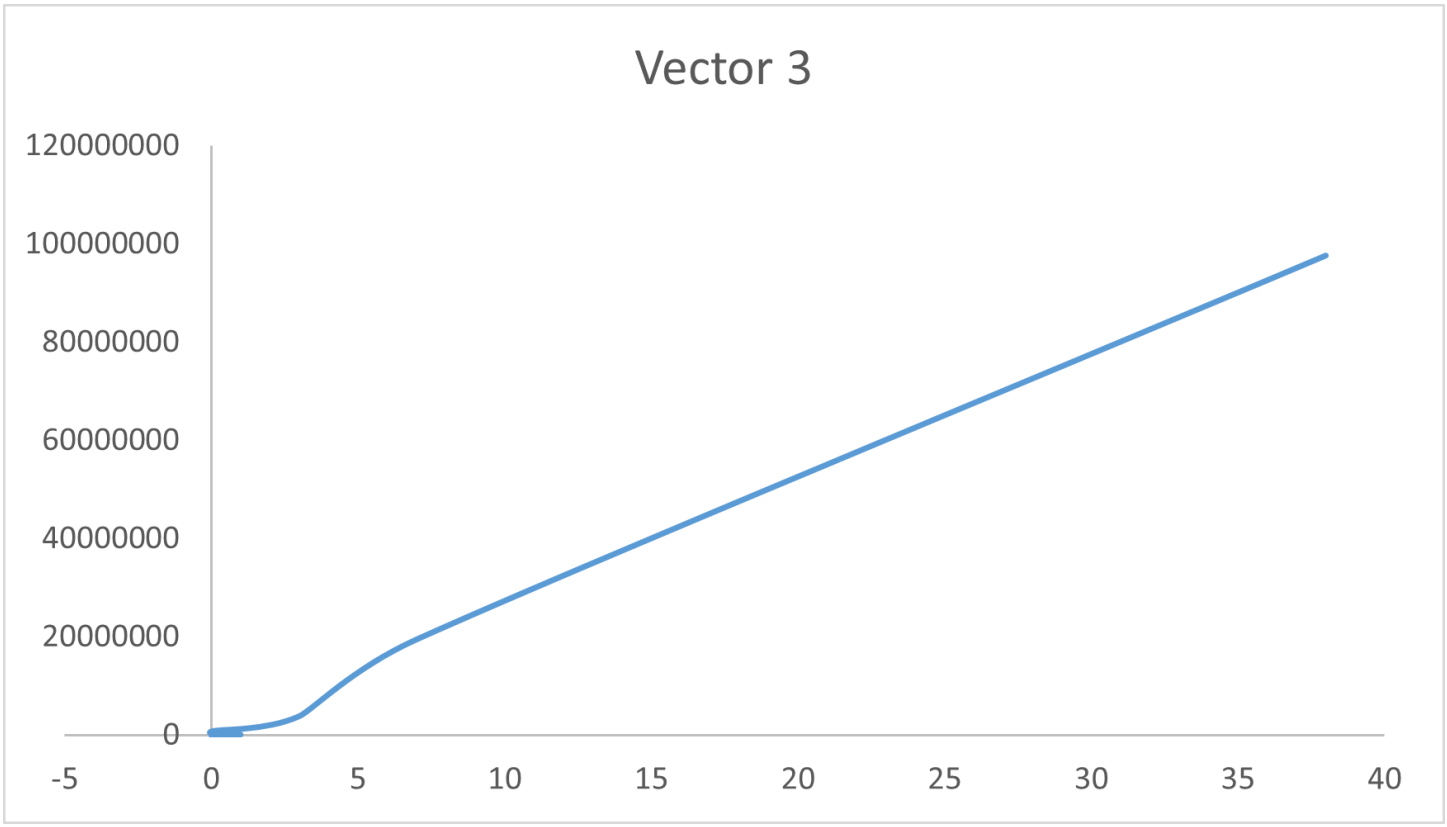
1.What happens with time if the size of the problem is multiplied by 5?

That the execution time of the problem increments for each new iteration, since n is 5 times bigger than before.

2. Are the times obtained those that were expected from linear complexity O(n)?

Yes, the times are the expected for a linear complexity.

3.Use a spreadsheet to draw a graph with Excel. On the X axis we can put the time and on the Y axis the size of the problem.



Taking small execution times

n	*fillIn(t)*	*sum(t)*	*maximum(t)*
10	20	0	2
30	1	0	0
90	2	0	0
270	3	0	0
310	3	0	0
810	10	0	0
2430	31	0	1
7290	85	3	0
21870	268	7	50
65610	755	23	75
196830	2186	70	322
590490	7816	204	962
1771470	31396	692	2594
Until it crashes	152259	25021	63243

Processor: Intel(R) Core (TM) i7-6700HQ CPU @ 2.60GHz 2.59 GHz

Memory: 8GB

The actual values are very close to the theoretical ones, due to its simple linear complexity.

	Theoretical	Actual
sum (n= 65610)	612	692
max(n=196830)	2886	2594
fillIn(n=1771470)	612	692

Operations on matrices

n	*sumDiagonal1(t)*	*sumDiagonal2(t)*
10	0	1
30	3	0
90	4	0
270	40	1
810	334	3
2430	2849	27
7290	23894	182
21870	127135	705

Processor: AMD RYZEN 5 3500X 6-Core Processor 3.6 GHz

Memory: 16 G

The actual values are bigger than the theoretical ones for the Diagonal1 method, probably because of its squared complexity.

But the values for the Diagonal2 are also a little bit bigger than the theoretical ones even if its complexity is linear, although it is closer than the ones for Diagonal1

	Theoretical	Actual
sumDiagonal1	1002	2849
sumDiagonal2	546	705

Benchmarking

1.Why do you get differences in execution time between the two programs?

This happens because each programming language manages the execution of their programs (the different operations and the compilation) in a different way.

2.Regardlss of the specific times, is there any analogy in the behavior of the two implementations?

Yes, python takes higher times and grows faster than java, so it can only reach half of the sizes that java can generate for the program’s execution.