Carla Fernández González. UO244965

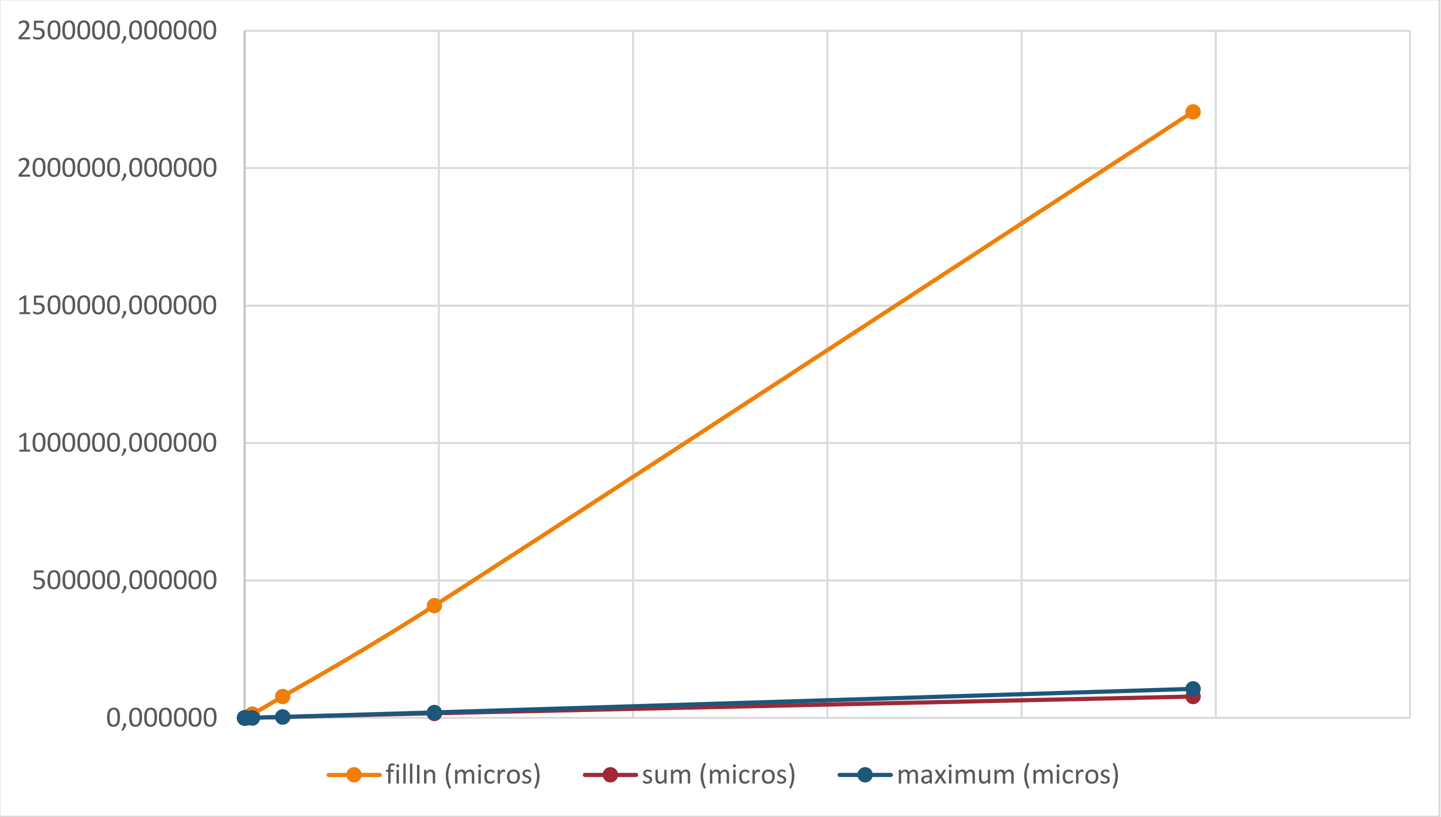
Algorithmics: Complexities and times

1 LAB 1

VECTOR 1 MEASUREMENTS*:*

The following measurements corresponds to the different methods in *Vector1*.

n	fillIn (micros)	sum (micros)	maximum (micros)
10	0,3100	0,0062	0,0124
50	1,1000	0,0343	0,0328
250	4,5200	0,2091	0,1638
1250	22,4600	0,9400	0,7349
6250	118,60	5,46	3,43
31250	680,80	26,98	18,10
156250	3061,30	137,13	88,46
781250	14040	811,05	759,63
3906250	78160	3450	3916
19531250	408890	17223	20200
97656250	2205570	78000	106100



As we can see, the fillIn() method takes a lot more time than the other two. The reason for this could be that the fillIn() method creates a new random number for each cell in the array, which takes a lot of resources.

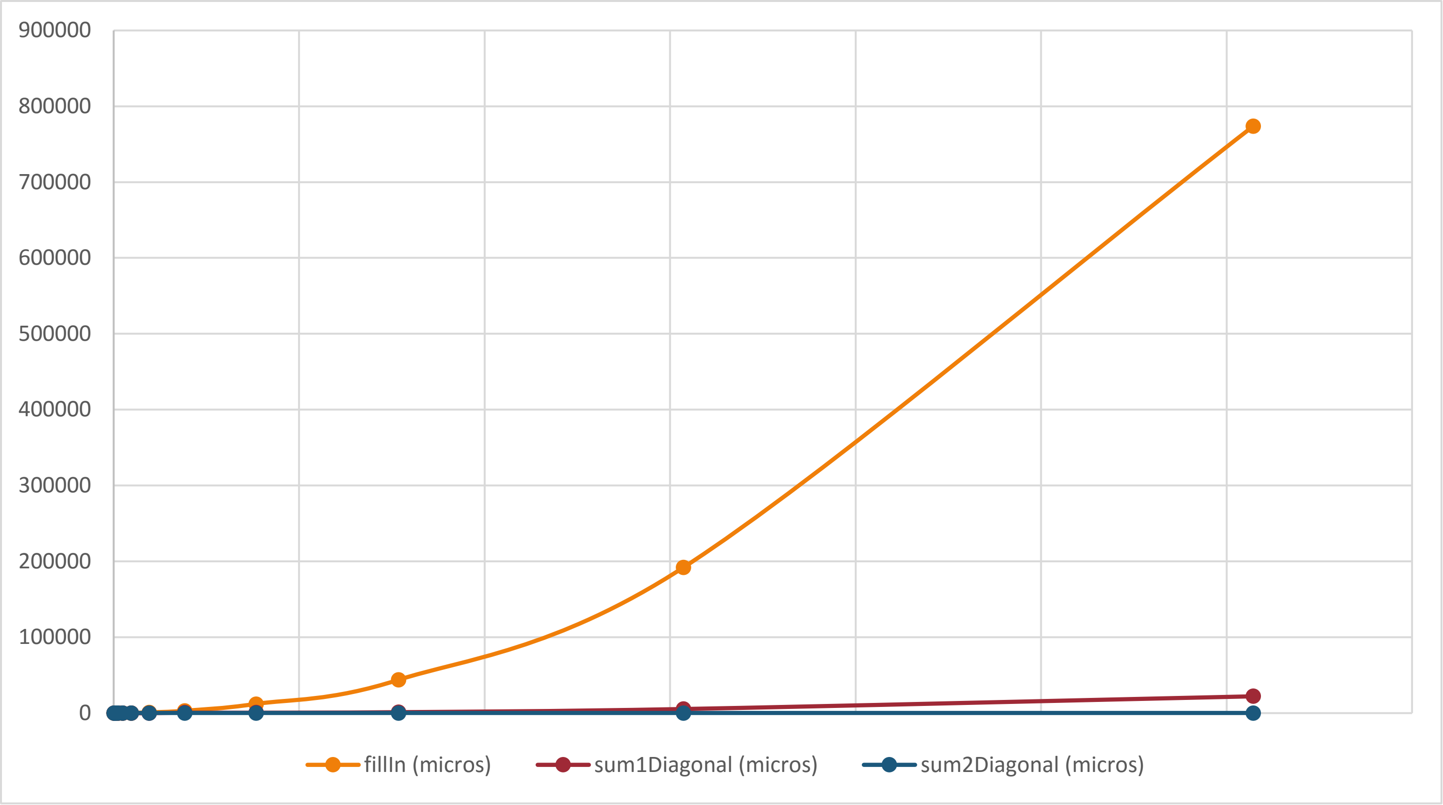
In conclusion, even though the three methods have an O(n) complexity (they iterate through the whole array), their runtime performance differs hugely due to the different operations carried out in each one.

DIAGONAL 1 MEASUREMENTS*:*

The following measurements corresponds to the different methods in *Diagonal1*.

Now we are dealing with a matrix and the results repeat themselves. The main difference now is that not all the methods have the same complexity. FillIn() and sum1Diagonal() have O(n^2) complexities, while sum2Diagonal() has O(n) complexity. This is the reason why sum1Diagonal() is slower than sum2Diagonal(). FillIn() is again much slower than the other two due to the random numbers it generates inside its loops.

n	fillIn (micros)	sum1Diagonal (micros)	sum2Diagonal (micros)
3	0,148	0,007947	0,002157
6	0,466	0,0322	0,004559
12	1,759	0,0965	0,0083
24	6,906	0,4011	0,0163
48	27,4	1,88	0,032
96	111,3	6,26	0,0682
192	442,6	21,54	0,1464
384	2050	77,74	0,5491
768	7220	288	1,14
1536	113660	1111	1,61
3072	457000	4429	4,809
6144	1825000	17800	12,2



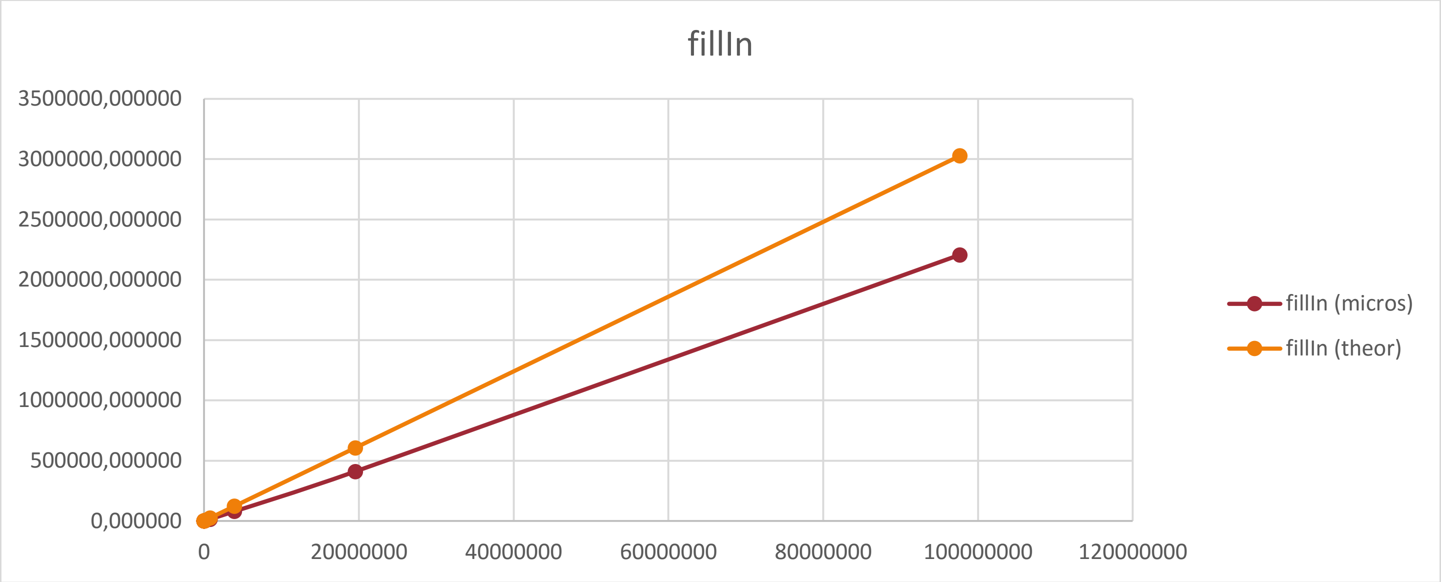
LAB1 CONCLUSSIONS:

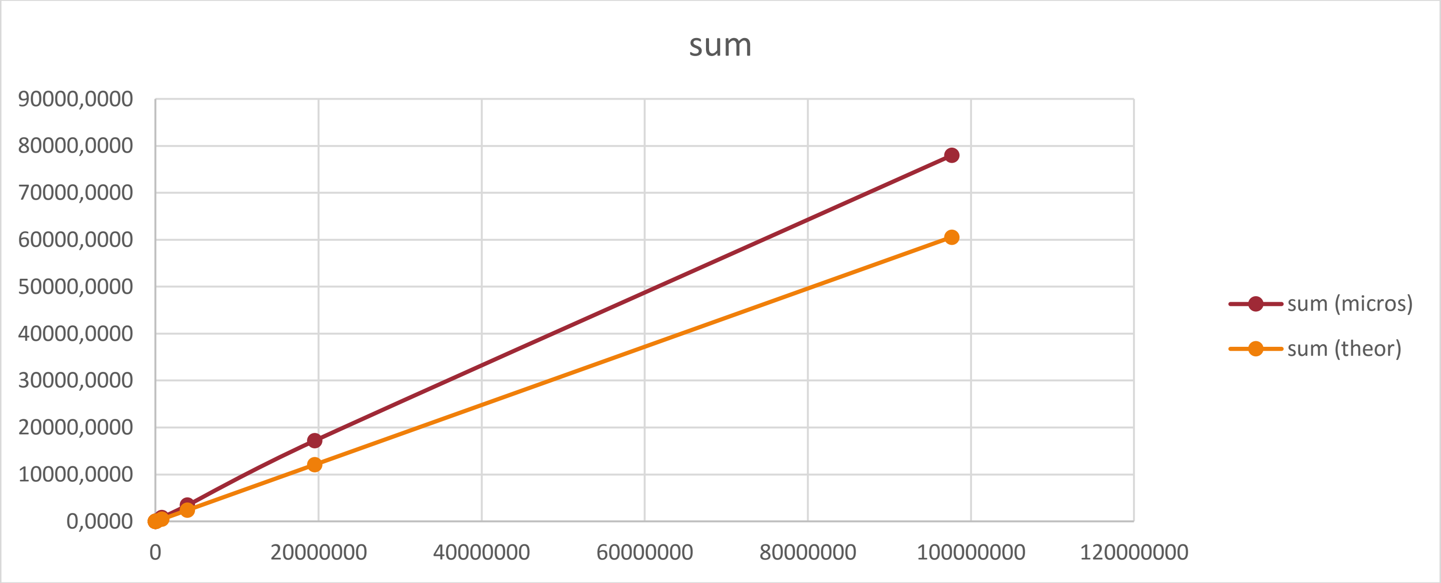
1) *Do the values obtained meet the expectations? For that, you should calculate and indicate the theoretical values of the complexity for all the methods.*

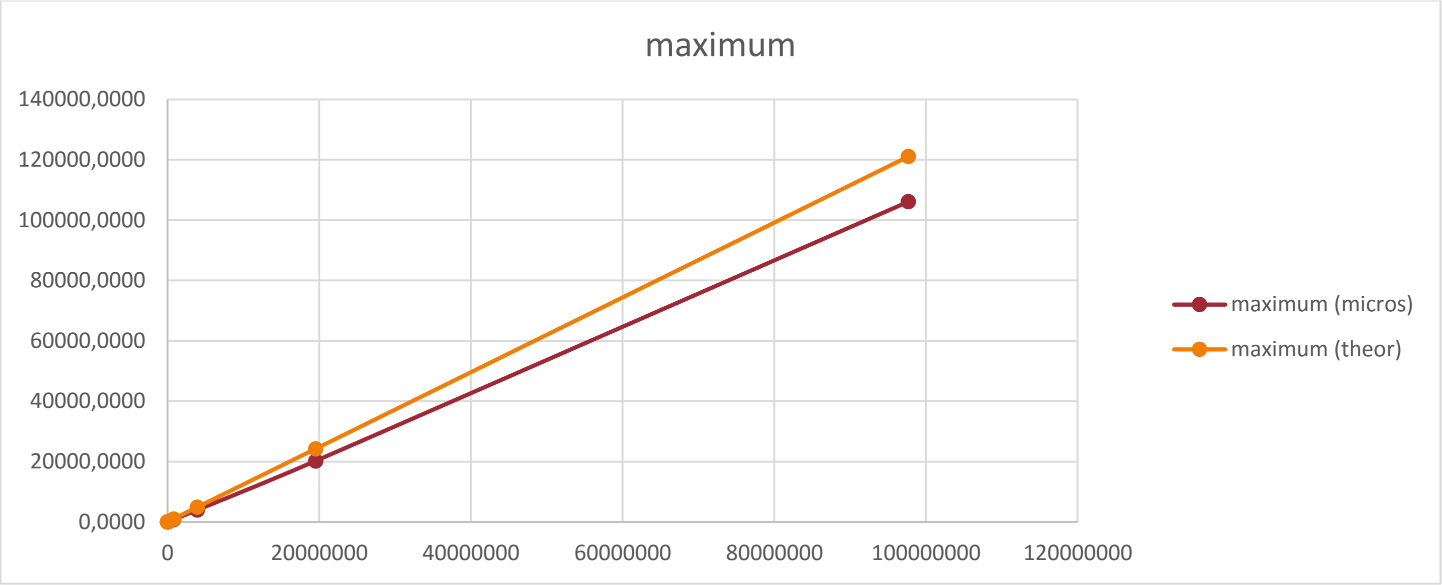
Vector1 theoretical values, based on the first measurement for n = 10, are the following:

n	fillIn (theor)	sum (theor)	maximum (theor)
10	0,31	0,01	0,01
50	1,55	0,03	0,06
250	7,75	0,16	0,31
1250	38,75	0,78	1,55
6250	193,75	3,88	7,75
31250	968,75	19,38	38,75
156250	4843,75	96,88	193,75
781250	24218,75	484,38	968,75
3906250	121093,75	2421,88	4843,75
19531250	605468,75	12109,38	24218,75
97656250	3027343,75	60546,88	121093,75

Compared with the actual times, we have this graphs:





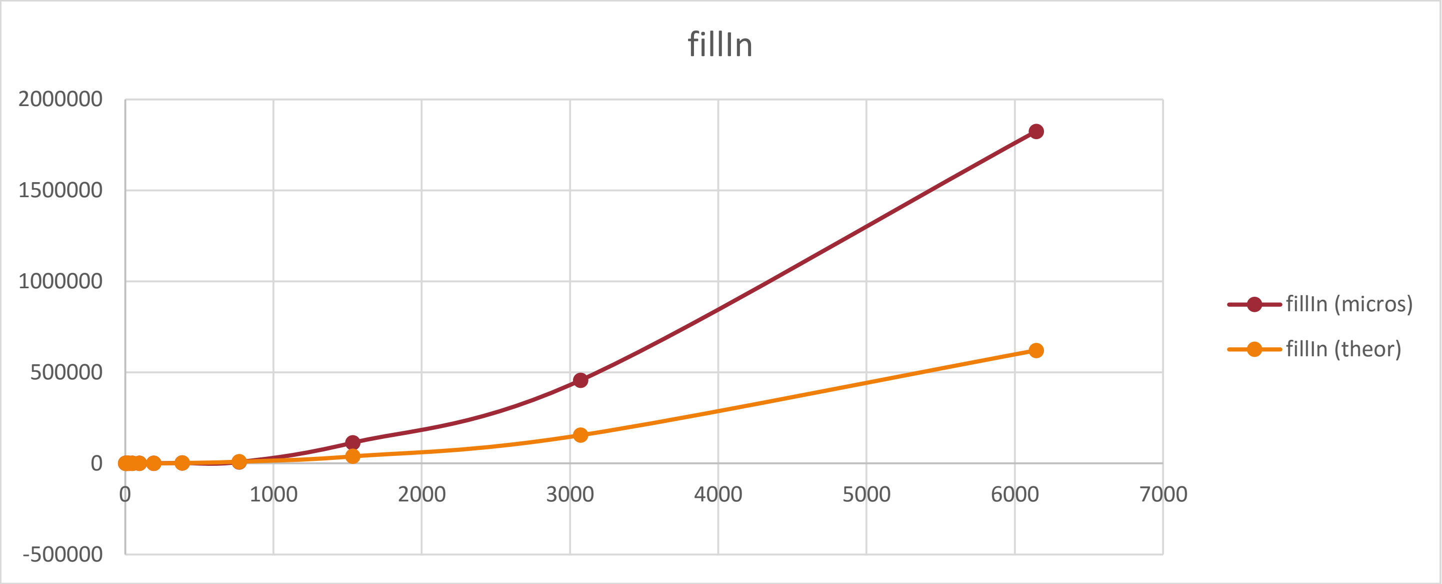


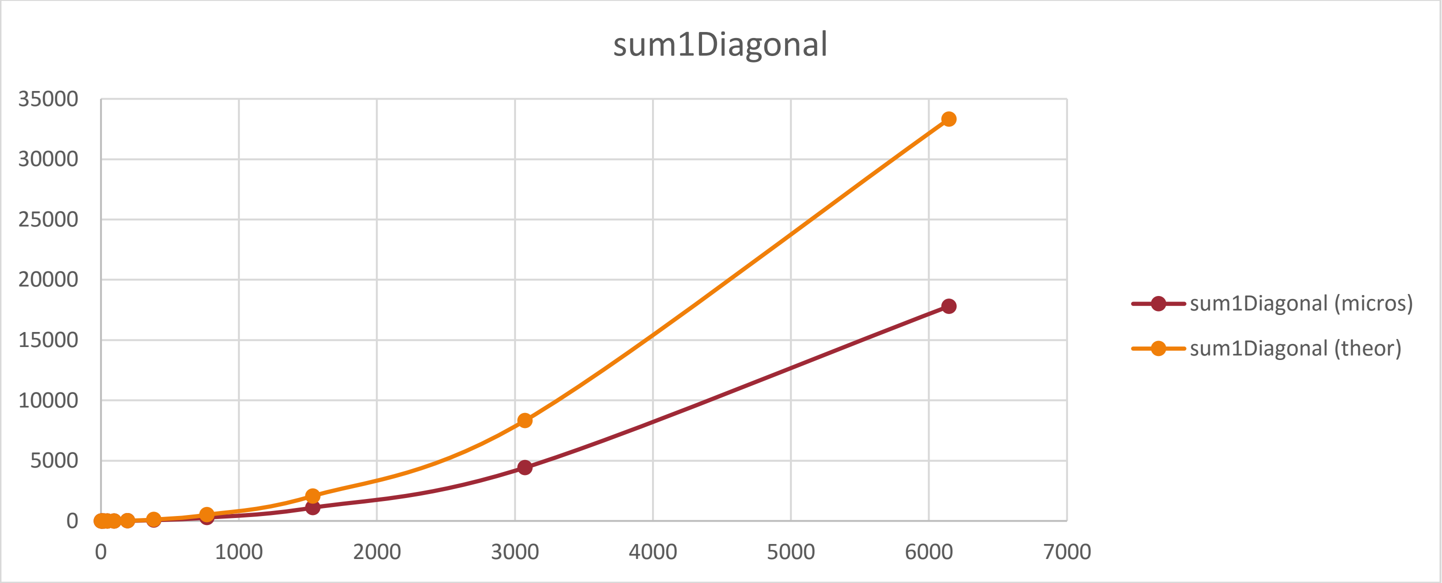
As we can see, in fillIn() and maximum() the theoretical times were higher than the actual times, while in sum() it was the other way around.

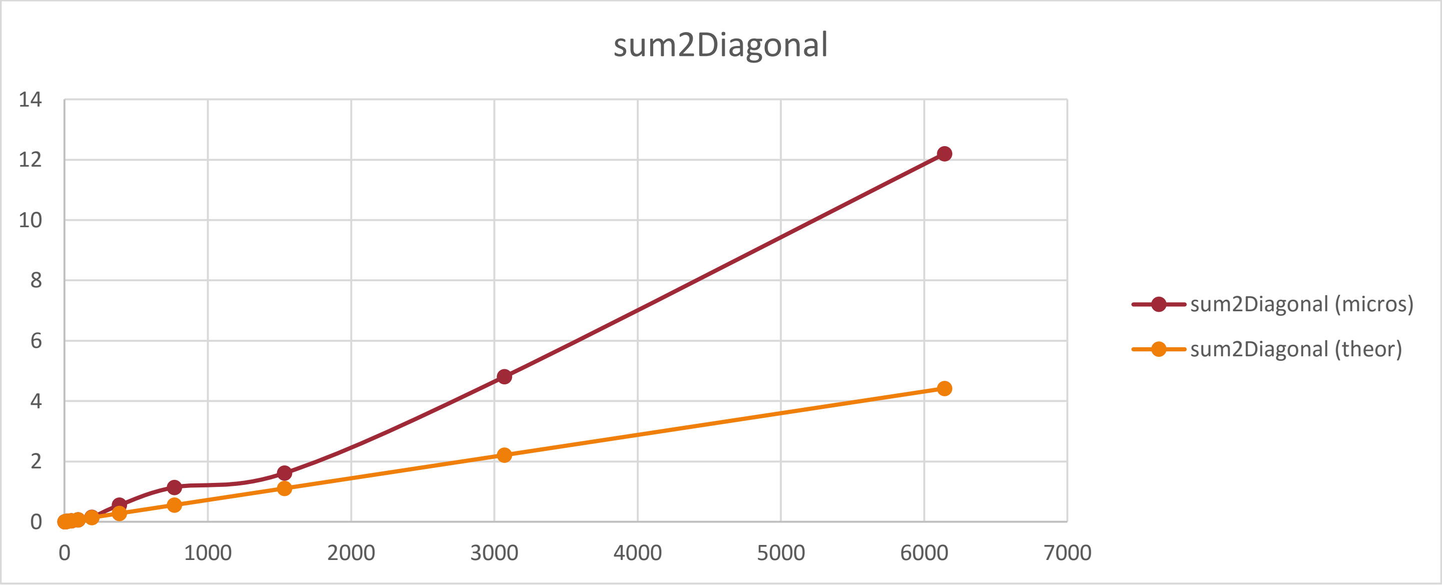
Diagonal1 theoretical values are the following:

n	fillIn (theor)	sum1Diagonal (theor)	sum2Diagonal (theor)
3	0,148	0,007947	0,002157
6	0,592	0,031788	0,004314
12	2,368	0,127152	0,008628
24	9,472	0,508608	0,017256
48	37,888	2,034432	0,034512
96	151,552	8,137728	0,069024
192	606,208	32,550912	0,138048
384	2424,832	130,203648	0,276096
768	9699,328	520,814592	0,552192
1536	38797,312	2083,258368	1,104384
3072	155189,248	8333,033472	2,208768
6144	620756,992	33332,13389	4,417536

Comparing with the actual times, we have these graphs:


fillIn
2000000
1500000
1000000
500000
0
-500000
0
1000
2000
3000
4000
5000
6000
7000
fillIn (micros)
fillIn (theor)


sum1Diagonal
35000
30000
25000
20000
15000
10000
5000
0
0
1000
2000
3000
4000
5000
6000
7000
sum1Diagonal (micros)
sum1Diagonal (theor)


sum2Diagonal
14
12
10
8
6
4
2
0
0
1000
2000
3000
4000
5000
6000
7000
sum2Diagonal (micros)
sum2Diagonal (theor)

As we can see, here the difference are much bigger and fillIn() and sum2Diagonal() perform worse than they should theoretically, while sum1Diagonal performs better.

2) *What are the main components of the computer in which you did the work?*

   Most of the work was done by the processor, in charge of doing the operations inside the loops. The memory also worked, but not as hardly.
   Out of curiosity, I ran maximum() from Vector1 while I watched over the Windows task manager. In my computer, the CPU usage increased from 1% (in rest) to 30%, and the memory usage only increased from 39% to 42%. Running the methods in Diagonal2, however, only raised my CPU up to 28% and didn't raise my memory usage at all.

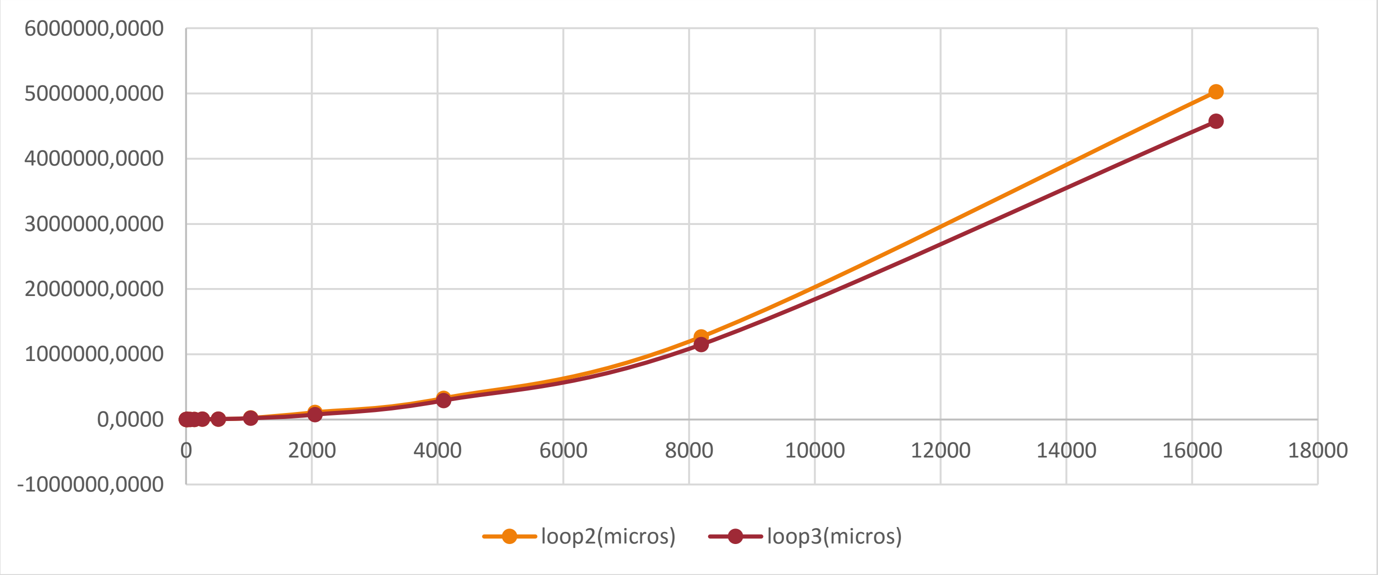
2 LAB 2

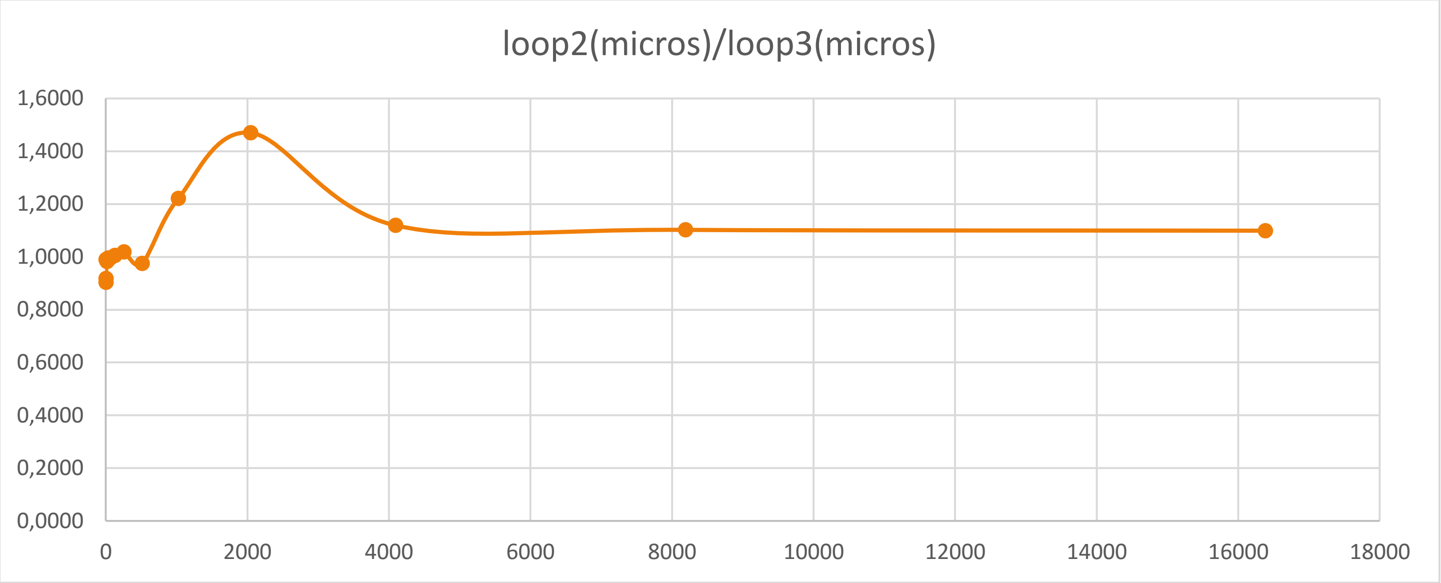
TABLE1: TWO ALGORITHMS WITH THE SAME COMPLEXITY

This table shows the values obtained for the executions of loop2 and loop3, both O(n^2) algorithms.

n	loop2(micros)	loop3(micros)	loop2(micros)/loop3(micros)
1	0,0930	0,0940	0,9894
2	0,1410	0,1560	0,9038
4	358,0000	390,0000	0,9179
8	1,1550	1,1650	0,9914
16	4,4780	4,5590	0,9822
32	17,6480	17,7100	0,9965
64	0,0699	0,0702	0,9957
128	287	285	1,0060
256	1134	1113	1,0184
512	4432	4545	0,9751
1024	21800	17853	1,2211
2048	107900	73400	1,4700
4096	324800	290200	1,1192
8192	1265800	1148300	1,1023
16384	5026000	4572000	1,0993

The corresponding graphs are:





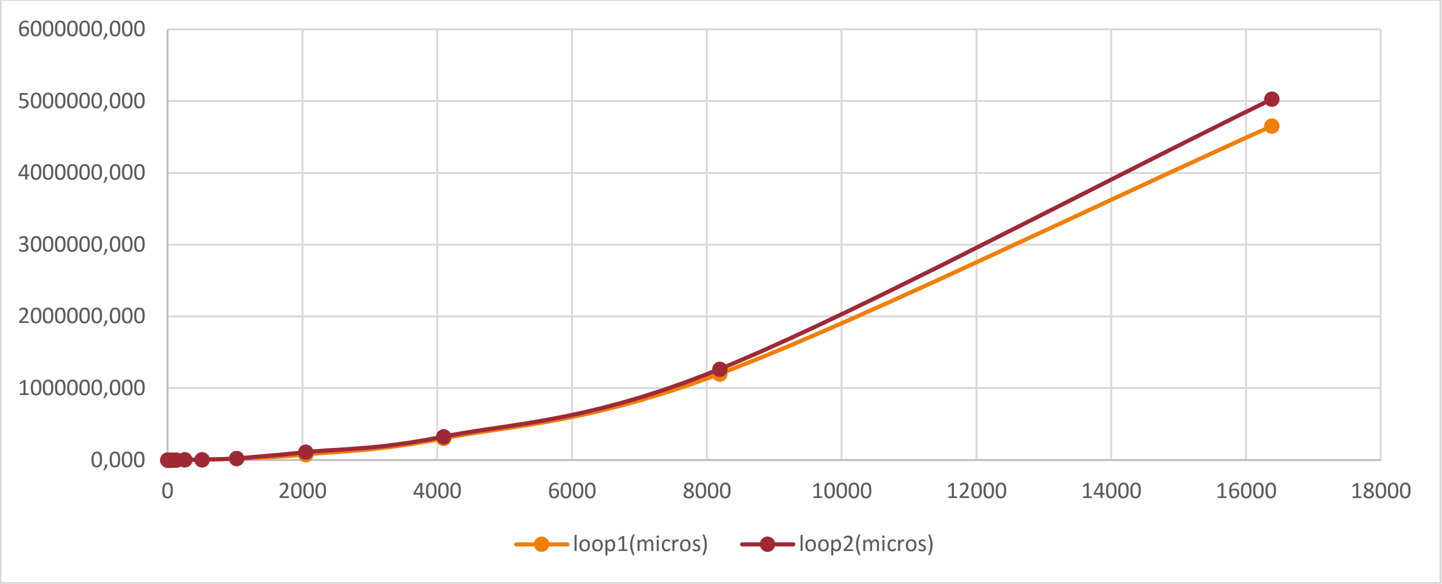
The results make sense with the complexity of the loops.The graph shows quadratic functions both for loop2 and loop3. By dividing loop2/loop3 times, we get the implementation constant, which tells us which loop performs better given n. Loop3 is better from n = [1024,4096], but then both loops perform more or less equally.

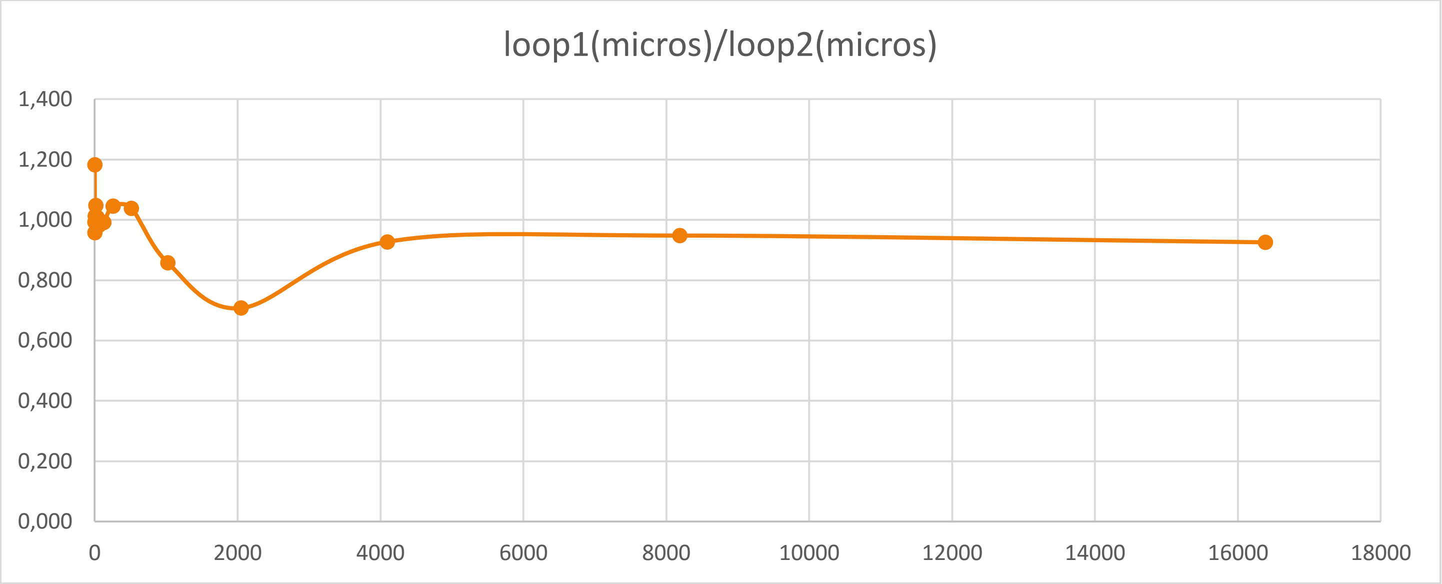
<u>TABLE2: TWO ALGORITHMS WITH DIFFERENT COMPLEXITY</u>

Now we compare an O(n logn) algorithm, loop1, with an O(n^2) algorithm, loop2.

n	loop1(micros)	loop2(micros)	loop1(micros)/loop2(micros)
1	0,110	0,093	1,183
2	0,140	0,141	0,993
4	0,343	0,358	0,958
8	1,170	1,155	1,013
16	4,692	4,478	1,048
32	17,806	17,648	1,009
64	68,700	69,900	0,983
128	284,400	286,700	0,992
256	1185	1133,500	1,045
512	4604	4432	1,039
1024	18700	21800	0,858
2048	76400	107900	0,708
4096	301200	324800	0,927
8192	1199900	1265800	0,948
16384	4652000	5026000	0,926

The graphs are:



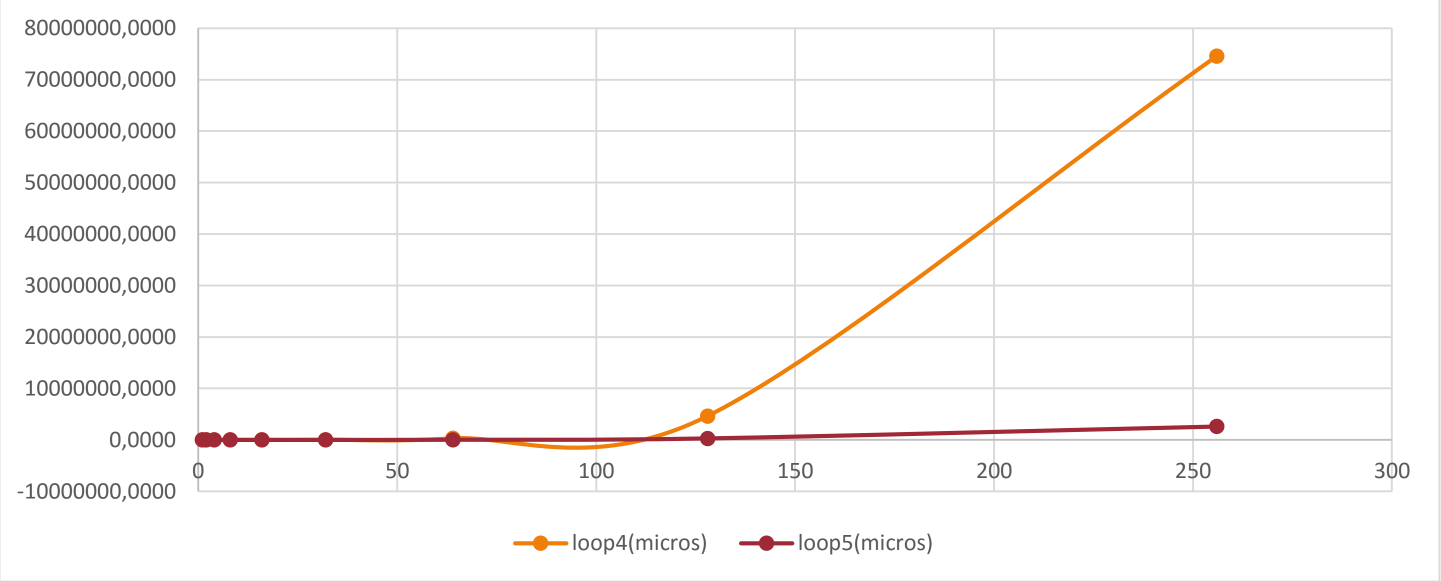


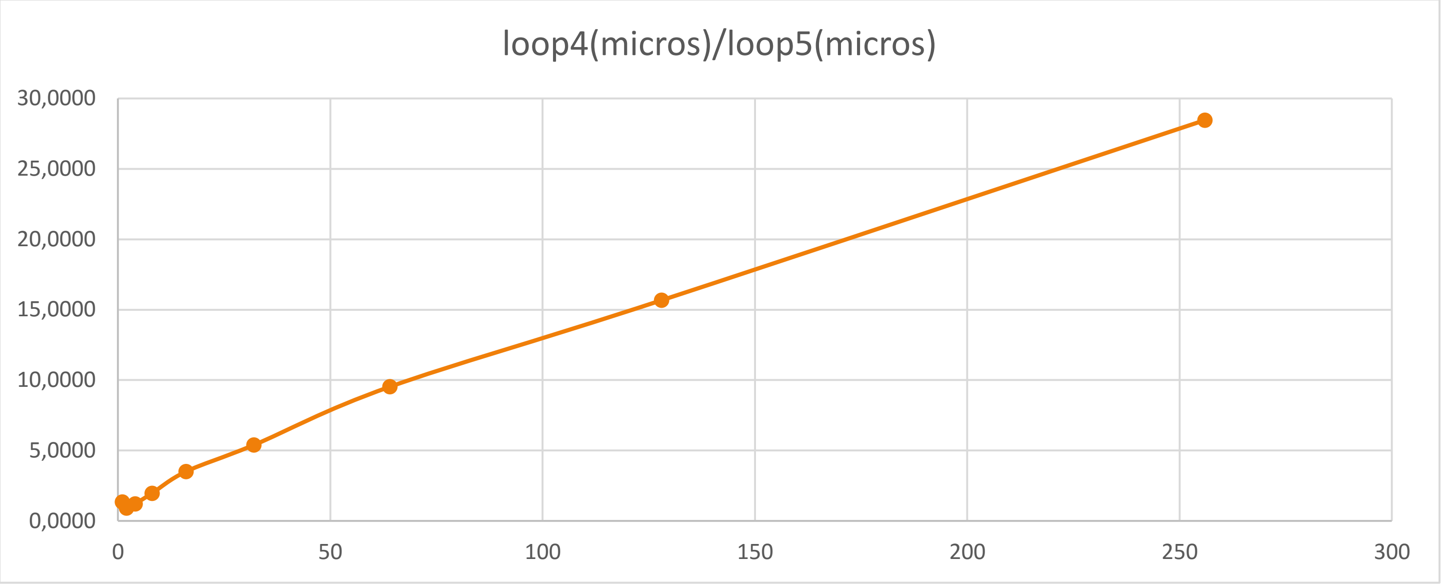
Here, the difference between one algorithm and the other does not seem too obvious, but in the table we can see how with big numbers (n > 16384) the difference would grow bigger and bigger. O(n logn) would show itself faster than O(n^2). For n = [1024, 2048] the implementation constant shows that loop1 is much faster.

TABLE3 == TWO ALGORITHMS WITH DIFFERENT COMPLEXITY:

I was asked to create loop4, an O(n^4) algorithm, and loop5, an O(n^3) algorithm and obtain the following table and graphs:

n	loop4(micros)	loop5(micros)	loop4(micros)/loop5(micros)
1	0,1250	0,0940	1,3298
2	0,3430	0,3740	0,9171
4	4,5270	3,7600	1,2040
8	70,2	35,9	1,9554
16	1250	357,2	3,4994
32	18410	3417	5,3878
64	297000	31200	9,5192
128	4617000	294800	15,6615
256	74585000	2621000	28,4567





The results are smashing: loop4 rises way higher than loop5 even with small numbers of n (n > 128). This proves how important algorithm complexity is, and just how bad an O(n^4) complexity is. The implementation constant is also clear: for every value, loop5 performs better than loop4.

Extra question: *What is the complexity of the method contained in Unknown? Why?*

The complexity of unknowned() is O(n^3) because it has three nested loops, each of which is proportional to n.