ALGORITMIA

PRÁCTICA 3

Héctor Lavandeira Fernández
UO277303 | UNIVERSIDAD DE OVIEDO – CURSO 2021/22

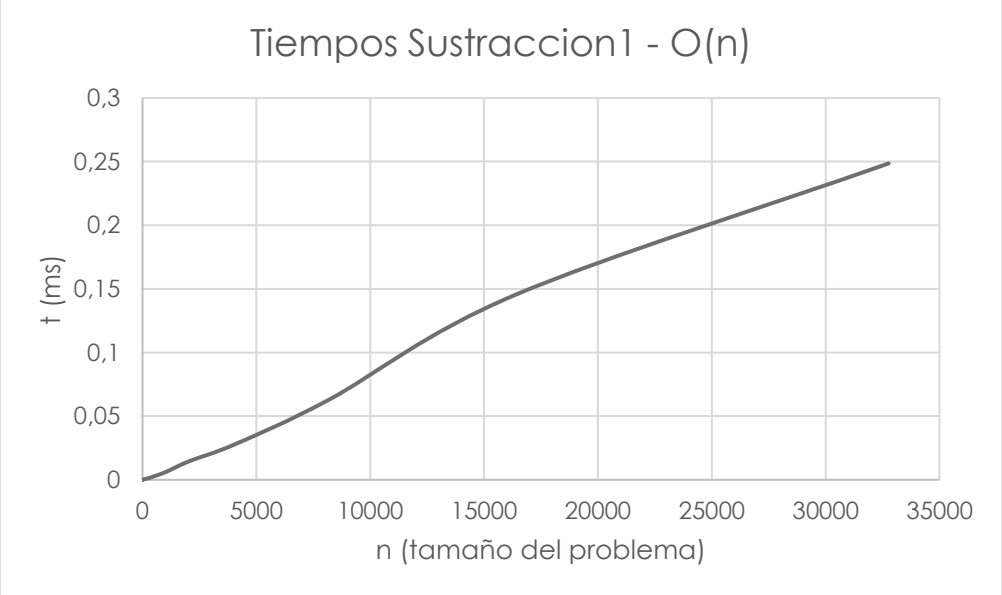
Características principales del ordenador:

Procesador:	i5-8250U
Memoria RAM:	8GB

SUSTRACCIÓN

Mediciones de tiempo de **Sustraccion1**:

n	*t_sustraccion1 (ms)*	*repeticiones*	*t / repeticiones (ms)*	*t_calculado*	*constante*
1	28	5000000	0,0000056		
2	118	5000000	0,0000236	0,0000112	2
4	141	5000000	0,0000282	0,0000472	2
8	230	5000000	0,000046	0,0000564	2
16	404	5000000	0,0000808	0,000092	2
32	766	5000000	0,0001532	0,0001616	2
64	1771	5000000	0,0003542	0,0003064	2
128	3208	5000000	0,0006416	0,0007084	2
256	6074	5000000	0,0012148	0,0012832	2
512	13905	5000000	0,002781	0,0024296	2
1024	30591	5000000	0,0061182	0,005562	2
2048	1454	100000	0,01454	0,0122364	2
4096	2817	100000	0,02817	0,02908	2
8192	6295	100000	0,06295	0,05634	2
16384	14537	100000	0,14537	0,1259	2
32768	24855	100000	0,24855	0,29074	2

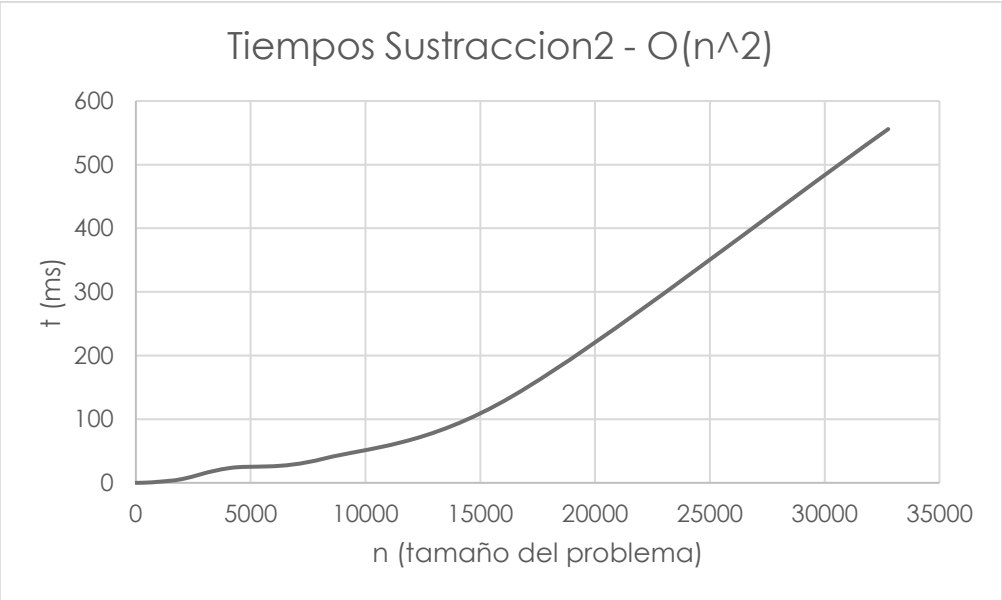


Observando la gráfica, podemos ver como tiene una tendencia lineal, es decir, el algoritmo tiene una complejidad O(n). Para asegurar esto, calculo los tiempos de la columna *t_calculado* mediante la fórmula de tiempos y tamaños vista en teoría. Al final, obtengo una

constante igual a 2 al comparar los tiempos obtenidos con los medidos. Esto significa que el algoritmo tiene la complejidad prevista.

Mediciones de tiempo de **Sustraccion2**:

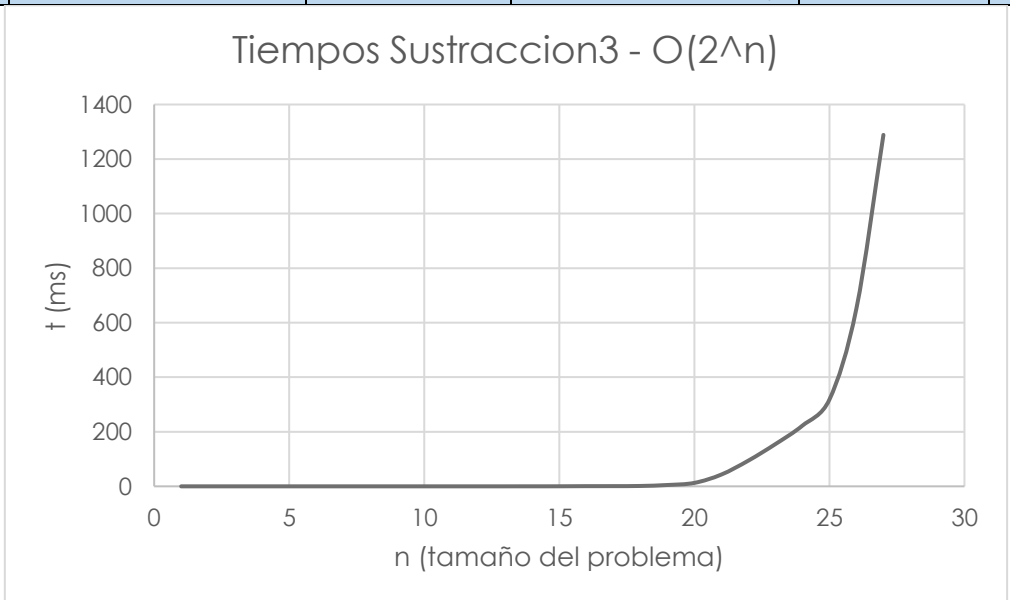
n	*t_sustraccion2 (ms)*	*repeticiones*	*t / repeticiones (ms)*	*t_calculado*	*constante*
1	60	1000000	0,00006		
2	85	1000000	0,000085	0,00024	4
4	134	1000000	0,000134	0,00034	4
8	290	1000000	0,00029	0,000536	4
16	714	1000000	0,000714	0,00116	4
32	2385	1000000	0,002385	0,002856	4
64	9141	1000000	0,009141	0,00954	4
128	37645	1000000	0,037645	0,036564	4
256	13937	100000	0,13937	0,15058	4
512	46013	100000	0,46013	0,55748	4
1024	1744	1000	1,744	1,84052	4
2048	6148	1000	6,148	6,976	4
4096	23295	1000	23,295	24,592	4
8192	3806	100	38,06	93,18	4
16384	13582	100	135,82	152,24	4
32768	55591	100	555,91	543,28	4



Este algoritmo, según los tiempos representados, podemos observar como no tiene una complejidad lineal. Al calcular los tiempos mediante la fórmula, y compararlos con los tiempos obtenidos, obtenemos una constante de 4. Esto nos permite confirmar que la complejidad del algoritmo es cuadrática, es decir, $O(n^2)$.

Mediciones de tiempo de **Sustraccion3**:

n	*t_sustraccion3(ms)*	*repeticiones*	*t / repeticiones (ms)*	*t_calculado*	*constante*
1	37	1000000	0,000037		
2	57	1000000	0,000057	0,000074	2
3	107	1000000	0,000107	0,000114	2
4	191	1000000	0,000191	0,000214	2
5	222	1000000	0,000222	0,000382	2
6	782	1000000	0,000782	0,000444	2
7	1158	1000000	0,001158	0,001564	2
8	3169	1000000	0,003169	0,002316	2
9	5643	1000000	0,005643	0,006338	2
10	10803	1000000	0,010803	0,011286	2
11	17908	1000000	0,017908	0,021606	2
12	42481	1000000	0,042481	0,035816	2
13	59364	1000000	0,059364	0,084962	2
14	1391	10000	0,1391	0,118728	2
15	2018	10000	0,2018	0,2782	2
16	6745	10000	0,6745	0,4036	2
17	7636	10000	0,7636	1,349	2
18	139	100	1,39	1,5272	2
19	490	100	4,9	2,78	2
20	1267	100	12,67	9,8	2
21	4330	100	43,3	25,34	2
22	9436	100	94,36	86,6	2
23	15490	100	154,9	188,72	2
24	22268	100	222,68	309,8	2
25	31807	100	318,07	445,36	2
26	6525	10	652,5	636,14	2
27	12886	10	1288,6	1305	2



Al representar los tiempos de este algoritmo, vemos que la gráfica crece aún más rápido que en el caso anterior (complejidad cuadrática). Por esto, vamos a analizar el código para intentar calcular la complejidad:

```
public static boolean rec3(int n) {
    if (n <= 0)
        cont++;
    else {
        cont++;
        rec3(n - 1);
        rec3(n - 1);
    }
    return true;
}
```

En este método hay 2 llamadas recursivas (es decir, a = 2).

El parámetro del método se reduce restando 1 (es decir, b = 1).

Y el resto del método, tiene una complejidad O(1), es decir, k = 1.

Con estos datos, la complejidad obtenida es $O(2^n)$.

Tras calcular los tiempos utilizando la fórmula y compararlos con los obtenidos en la medición, obtenemos una constante de 2, por lo que podemos confirmar que la complejidad del método es $O(2^n)$.

DIVISION

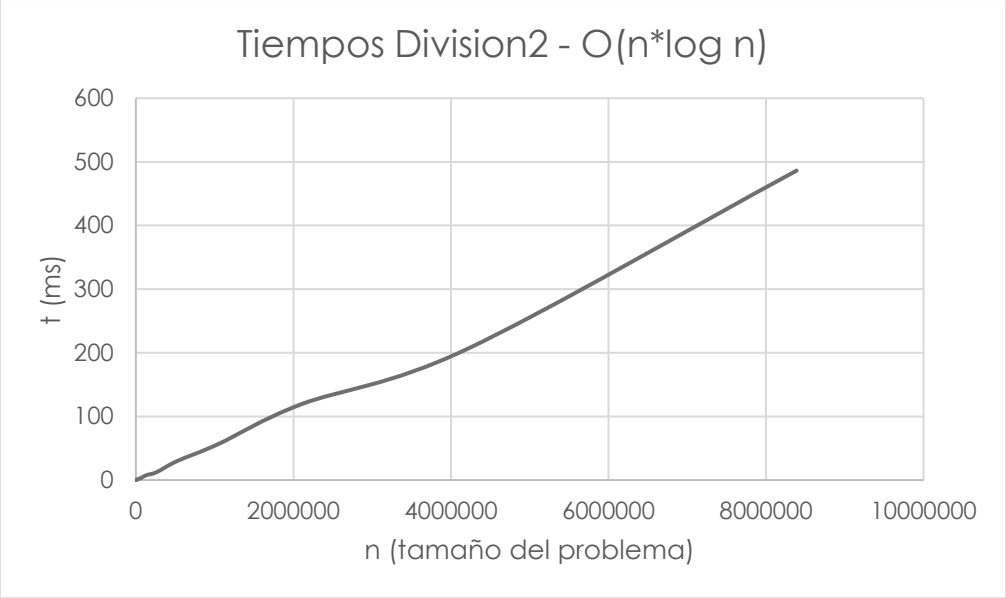
Mediciones de tiempo para **Division1**:

n	*t_division1 (ms)*	*repeticiones*	*t / repeticiones (ms)*	*t_calculado*	*constante*
1	37	10000000	0,0000037		
2	179	10000000	0,0000179	0,0000074	2
4	212	10000000	0,0000212	0,0000358	2
8	410	10000000	0,000041	0,0000424	2
16	609	10000000	0,0000609	0,000082	2
32	1151	10000000	0,0001151	0,0001218	2
64	2051	10000000	0,0002051	0,0002302	2
128	3800	10000000	0,00038	0,0004102	2
256	7602	10000000	0,0007602	0,00076	2
512	14323	10000000	0,0014323	0,0015204	2
1024	28256	10000000	0,0028256	0,0028646	2
2048	56235	10000000	0,0056235	0,0056512	2
4096	115571	10000000	0,0115571	0,011247	2
8192	2524	100000	0,02524	0,0231142	2
16384	4776	100000	0,04776	0,05048	2
32768	9410	100000	0,0941	0,09552	2
65536	18962	100000	0,18962	0,1882	2
131072	40533	100000	0,40533	0,37924	2
262144	80665	100000	0,80665	0,81066	2
524288	976	1000	0,976	1,6133	2
1048576	1948	1000	1,948	1,952	2
2097152	3597	1000	3,597	3,896	2
4194304	6997	1000	6,997	7,194	2
8388608	14339	1000	14,339	13,994	2

La complejidad de este algoritmo es O(n), ya que coincide la representación de los tiempos, la comparación con los tiempos obtenidos mediante la fórmula, y el análisis de la complejidad a partir del código.

Mediciones de tiempo para **Division2**:

n	*t_division2 (ms)*	*repeticiones*	*t / repeticiones (ms)*	*t_calculado*	*constante*
1	31	10000000	0,0000031		
2	410	10000000	0,000041		
4	653	10000000	0,0000653	0,000164	4
8	2112	10000000	0,0002112	0,0001959	3
16	3238	10000000	0,0003238	0,0005632	2,666666667
32	9520	10000000	0,000952	0,0008095	2,5
64	15156	10000000	0,0015156	0,0022848	2,4
128	43945	10000000	0,0043945	0,0035364	2,333333333
256	126	10000	0,0126	0,010044571	2,285714286
512	267	10000	0,0267	0,02835	2,25
1024	475	10000	0,0475	0,059333333	2,222222222
2048	1096	10000	0,1096	0,1045	2,2
4096	1759	10000	0,1759	0,239127273	2,181818182
8192	4299	10000	0,4299	0,381116667	2,166666667
16384	7210	10000	0,721	0,925938462	2,153846154
32768	18331	10000	1,8331	1,545	2,142857143
65536	31836	10000	3,1836	3,910613333	2,133333333
131072	760	100	7,6	6,76515	2,125
262144	1227	100	12,27	16,09411765	2,117647059
524288	2986	100	29,86	25,90333333	2,111111111
1048576	5662	100	56,62	62,86315789	2,105263158
2097152	11900	100	119	118,902	2,1
4194304	20521	100	205,21	249,3333333	2,095238095
8388608	48624	100	486,24	429,0754545	2,090909091

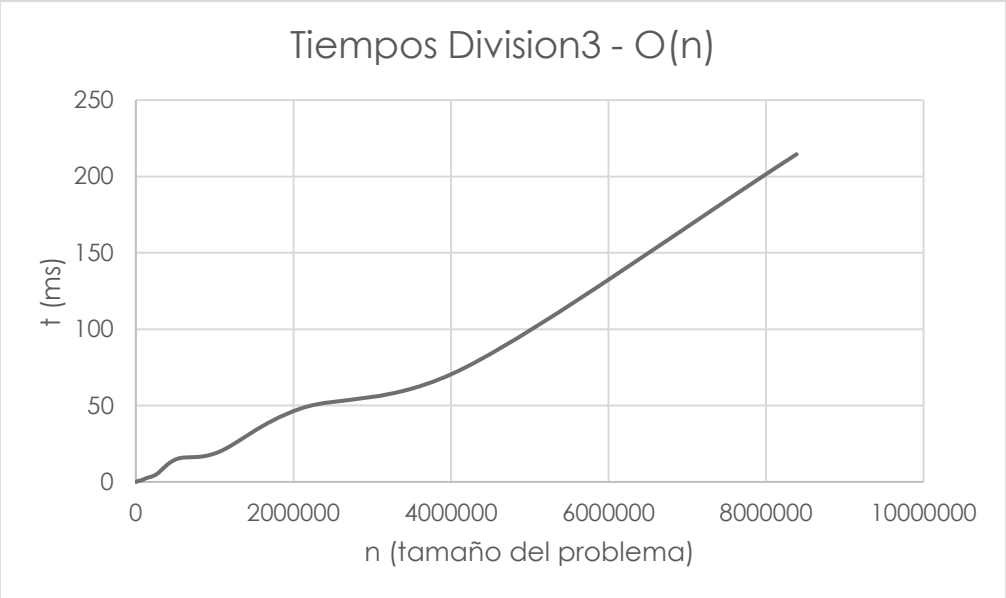


Observando la gráfica de este algoritmo, a simple vista puede parecer que su complejidad es lineal. Pero analizando el código, sabemos que la complejidad es O(n·log(n)). Al calcular los tiempos mediante la fórmula vista en teoría, y compararlos con los tiempos obtenidos, no se obtiene una constante, aunque la mayor parte de los valores se encuentran cerca de 2.

Mediciones de tiempo para **Division3**:

n	*t_division3 (ms)*	*repeticiones*	*t / repeticiones (ms)*	*t_calculado*	*constante*
1	31	10000000	0,0000031		
2	480	10000000	0,000048	0,0000062	2
4	519	10000000	0,0000519	0,000096	2
8	1817	10000000	0,0001817	0,0001038	2
16	1801	10000000	0,0001801	0,0003634	2
32	5626	10000000	0,0005626	0,0003602	2
64	8732	10000000	0,0008732	0,0011252	2
128	31550	10000000	0,003155	0,0017464	2
256	38583	10000000	0,0038583	0,00631	2
512	109746	10000000	0,0109746	0,0077166	2
1024	111	10000	0,0111	0,0219492	2
2048	345	10000	0,0345	0,0222	2
4096	598	10000	0,0598	0,069	2
8192	2434	10000	0,2434	0,1196	2
16384	2579	10000	0,2579	0,4868	2
32768	7272	10000	0,7272	0,5158	2
65536	9819	10000	0,9819	1,4544	2
131072	245	100	2,45	1,9638	2
262144	501	100	5,01	4,9	2
524288	1509	100	15,09	10,02	2
1048576	1951	100	19,51	30,18	2
2097152	4813	100	48,13	39,02	2

4194304	7521	100	75,21	96,26	2
8388608	21456	100	214,56	150,42	2



Analizando el código, la complejidad de este método es lineal, aunque al representar los tiempos en la gráfica salen algunos picos en varios valores de tiempo. Sin embargo, al calcular y comparar los tiempos calculados con los obtenidos, obtenemos una constante de 2.

FIBONACCI

Mediciones de tiempo para la **primera versión** de Fibonacci:

n	*t_fib1 (ms)*	*repeticiones*	*t / repeticiones (ms)*
10	75	4000000	0,00001875
11	79	4000000	0,00001975
12	45	4000000	0,00001125
13	44	4000000	0,000011
14	45	4000000	0,00001125
15	44	4000000	0,000011
16	64	4000000	0,000016
17	65	4000000	0,00001625
18	63	4000000	0,00001575
19	66	4000000	0,0000165
20	62	4000000	0,0000155
21	65	4000000	0,00001625
22	61	4000000	0,00001525
23	60	4000000	0,000015
24	56	4000000	0,000014
25	75	4000000	0,00001875
26	96	4000000	0,000024
27	100	4000000	0,000025

28	133	4000000	0,00003325
29	89	4000000	0,00002225
30	88	4000000	0,000022
31	113	4000000	0,00002825
32	87	4000000	0,00002175
33	115	4000000	0,00002875
34	126	4000000	0,0000315
35	127	4000000	0,00003175
36	134	4000000	0,0000335
37	109	4000000	0,00002725
38	108	4000000	0,000027
39	119	4000000	0,00002975
40	111	4000000	0,00002775
41	154	4000000	0,0000385
42	132	4000000	0,000033
43	116	4000000	0,000029
44	114	4000000	0,0000285
45	119	4000000	0,00002975
46	110	4000000	0,0000275
47	118	4000000	0,0000295
48	112	4000000	0,000028
49	134	4000000	0,0000335
50	149	4000000	0,00003725
51	151	4000000	0,00003775
52	134	4000000	0,0000335
53	138	4000000	0,0000345
54	152	4000000	0,000038
55	150	4000000	0,0000375
56	129	4000000	0,00003225
57	168	4000000	0,000042
58	190	4000000	0,0000475
59	165	4000000	0,00004125

Mediciones de tiempo para la **segunda versión** de Fibonacci:

n	*t_fib2 (ms)*	*repeticiones*	*t / repeticiones (ms)*
10	123	4000000	0,00003075
11	149	4000000	0,00003725
12	100	4000000	0,000025
13	112	4000000	0,000028
14	123	4000000	0,00003075
15	133	4000000	0,00003325
16	125	4000000	0,00003125
17	135	4000000	0,00003375
18	134	4000000	0,0000335

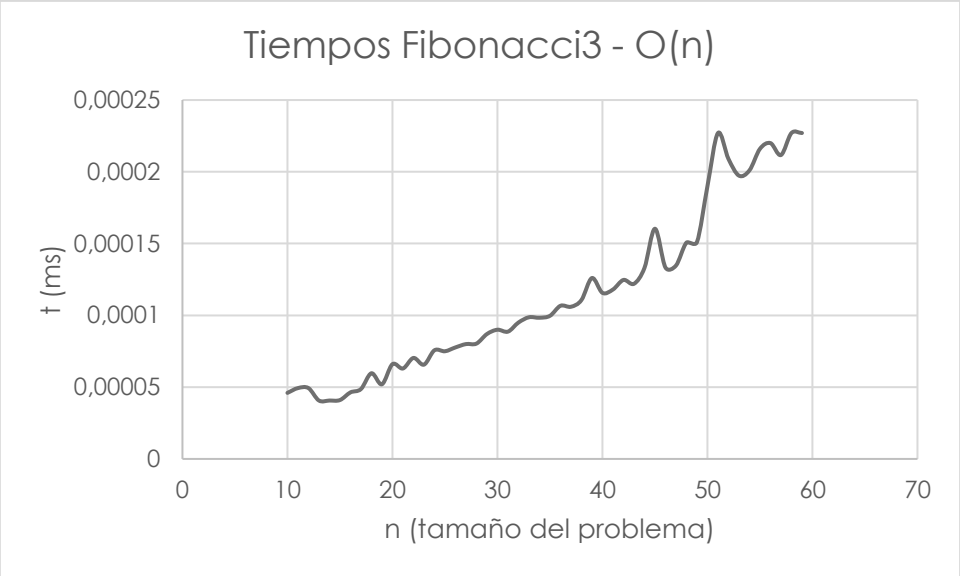
19	145	4000000	0,00003625
20	167	4000000	0,00004175
21	169	4000000	0,00004225
22	172	4000000	0,000043
23	180	4000000	0,000045
24	188	4000000	0,000047
25	202	4000000	0,0000505
26	218	4000000	0,0000545
27	216	4000000	0,000054
28	227	4000000	0,00005675
29	241	4000000	0,00006025
30	246	4000000	0,0000615
31	260	4000000	0,000065
32	264	4000000	0,000066
33	269	4000000	0,00006725
34	283	4000000	0,00007075
35	278	4000000	0,0000695
36	297	4000000	0,00007425
37	319	4000000	0,00007975
38	323	4000000	0,00008075
39	317	4000000	0,00007925
40	347	4000000	0,00008675
41	370	4000000	0,0000925
42	373	4000000	0,00009325
43	391	4000000	0,00009775
44	391	4000000	0,00009775
45	418	4000000	0,0001045
46	421	4000000	0,00010525
47	429	4000000	0,00010725
48	445	4000000	0,00011125
49	446	4000000	0,0001115
50	462	4000000	0,0001155
51	476	4000000	0,000119
52	473	4000000	0,00011825
53	471	4000000	0,00011775
54	476	4000000	0,000119
55	470	4000000	0,0001175
56	516	4000000	0,000129
57	543	4000000	0,00013575
58	528	4000000	0,000132
59	530	4000000	0,0001325

Estas dos versiones del algoritmo son iterativas. Ambas tienen una complejidad de O(n).

Mediciones de tiempo para la **tercera versión** de Fibonacci:

n	*t_fib3 (ms)*	*repeticiones*	*t / repeticiones (ms)*
10	138	3000000	0,000046
11	148	3000000	4,93333E-05
12	148	3000000	4,93333E-05
13	122	3000000	4,06667E-05
14	122	3000000	4,06667E-05
15	123	3000000	0,000041
16	139	3000000	4,63333E-05
17	146	3000000	4,86667E-05
18	179	3000000	5,96667E-05
19	156	3000000	0,000052
20	198	3000000	0,000066
21	189	3000000	0,000063
22	211	3000000	7,03333E-05
23	197	3000000	6,56667E-05
24	227	3000000	7,56667E-05
25	225	3000000	0,000075
26	233	3000000	7,76667E-05
27	240	3000000	0,00008
28	241	3000000	8,03333E-05
29	261	3000000	0,000087
30	270	3000000	0,00009
31	266	3000000	8,86667E-05
32	285	3000000	0,000095
33	296	3000000	9,86667E-05
34	295	3000000	9,83333E-05
35	299	3000000	9,96667E-05
36	320	3000000	0,000106667
37	318	3000000	0,000106
38	332	3000000	0,000110667
39	378	3000000	0,000126
40	347	3000000	0,000115667
41	354	3000000	0,000118
42	374	3000000	0,000124667
43	366	3000000	0,000122
44	399	3000000	0,000133
45	481	3000000	0,000160333
46	400	3000000	0,000133333
47	404	3000000	0,000134667
48	452	3000000	0,000150667
49	453	3000000	0,000151
50	570	3000000	0,00019
51	681	3000000	0,000227

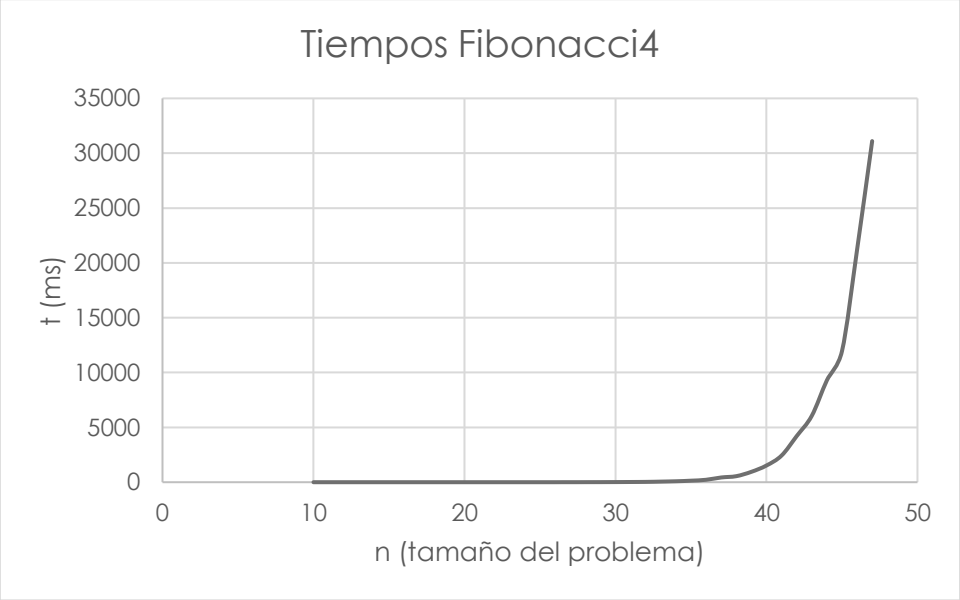
52	627	3000000	0,000209
53	592	3000000	0,000197333
54	603	3000000	0,000201
55	648	3000000	0,000216
56	660	3000000	0,00022
57	635	3000000	0,000211667
58	681	3000000	0,000227
59	681	3000000	0,000227



Mediciones de tiempo para la **cuarta versión** de Fibonacci:

n	*t_fib4 (ms)*	*repeticiones*	*t / repeticiones (ms)*
10	56	50000	0,00112
11	102	50000	0,00204
12	219	50000	0,00438
13	316	50000	0,00632
14	470	50000	0,0094
15	453	50000	0,00906
16	695	50000	0,0139
17	1031	50000	0,02062
18	1558	50000	0,03116
19	2557	50000	0,05114
20	3820	50000	0,0764
21	1832	10000	0,1832
22	2709	10000	0,2709
23	4497	10000	0,4497
24	7262	10000	0,7262
25	642	500	1,284
26	929	500	1,858
27	1301	500	2,602

28	2407	500	4,814
29	4052	500	8,104
30	6276	500	12,552
31	10240	500	20,48
32	14726	500	29,452
33	533	10	53,3
34	867	10	86,7
35	1384	10	138,4
36	2232	10	223,2
37	4312	10	431,2
38	5516	10	551,6
39	9470	10	947
40	15230	10	1523
41	24301	10	2430,1
42	42017	10	4201,7
43	60526	10	6052,6
44	92882	10	9288,2
45	11946	1	11946
46	21192	1	21192
47	31105	1	31105



Estas dos últimas versiones son recursivas. La tercera versión tiene una complejidad lineal, aunque al representar los tiempos en la gráfica, salen demasiados picos.

Por otra parte, la complejidad de la última versión no puede definirse exactamente, ya que las llamadas recursivas utilizan un parámetro modificado de forma diferente cada una: una resta 1 y otra resta 2.

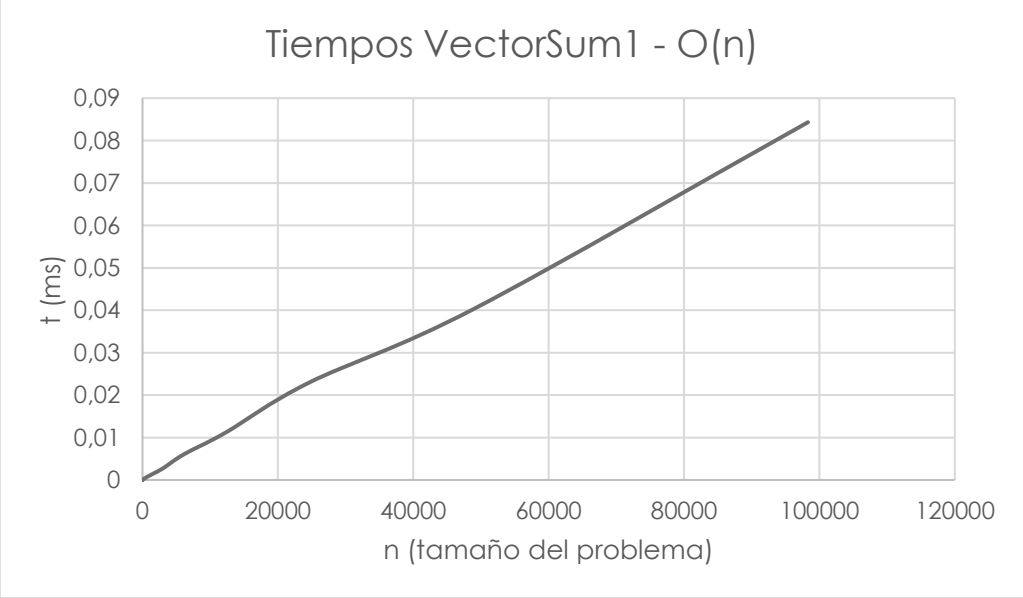
Al realizar los cálculos de complejidad, podemos asegurar que la complejidad del algoritmo se encuentra entre dos:

$$O(2^{n/2}) <= O(fib4) <= O(2^n)$$

VECTOR SUM

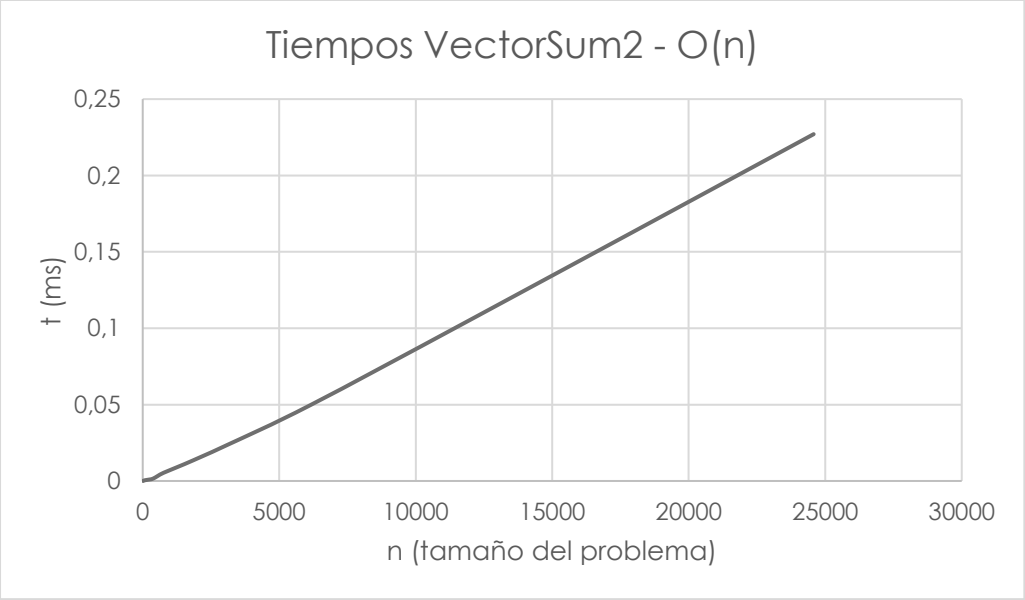
Mediciones de tiempo para la **primera versión**:

n	*t_sum1 (ms)*	*repeticiones*	*t / repeticiones (ms)*	*t_calculado*	*constante*
3	60	2500000	0,000024		
6	75	2500000	0,00003	0,000048	2
12	41	2500000	0,0000164	0,00006	2
24	81	2500000	0,0000324	0,0000328	2
48	165	2500000	0,000066	0,0000648	2
96	335	2500000	0,000134	0,000132	2
192	616	2500000	0,0002464	0,000268	2
384	1172	2500000	0,0004688	0,0004928	2
768	2007	2500000	0,0008028	0,0009376	2
1536	3654	2500000	0,0014616	0,0016056	2
3072	6963	2500000	0,0027852	0,0029232	2
6144	15192	2500000	0,0060768	0,0055704	2
12288	11173	1000000	0,011173	0,0121536	2
24576	22929	1000000	0,022929	0,022346	2
49152	40485	1000000	0,040485	0,045858	2
98304	8432	100000	0,08432	0,08097	2



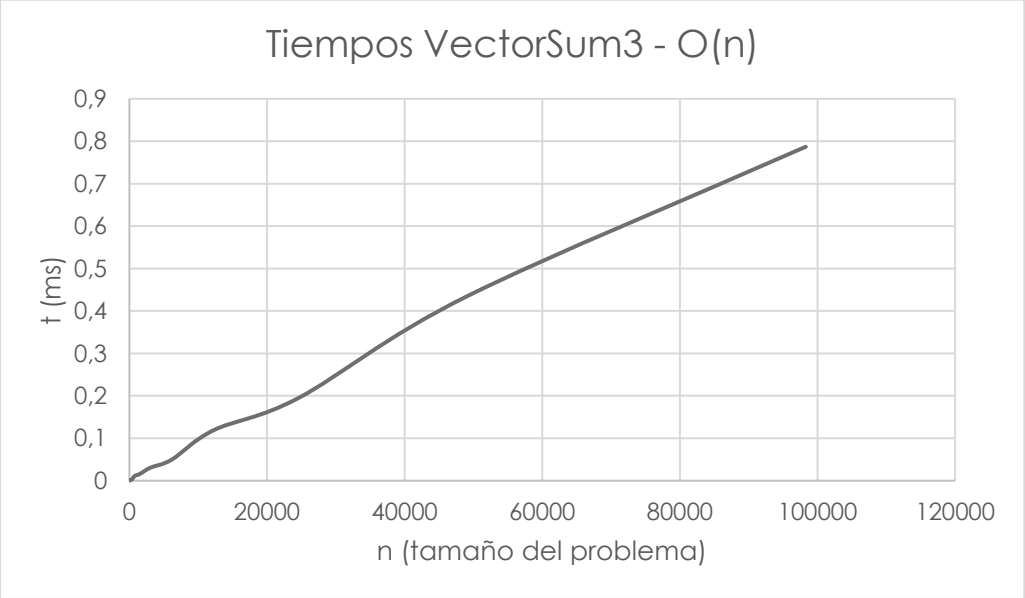
Mediciones de tiempo para la **segunda versión**:

n	*t_sum2 (ms)*	*repeticiones*	*t / repeticiones (ms)*	*t_calculado*	*constante*
3	55	1000000	0,000055		
6	68	1000000	0,000068	0,00011	2
12	53	1000000	0,000053	0,000136	2
24	120	1000000	0,00012	0,000106	2
48	276	1000000	0,000276	0,00024	2
96	528	1000000	0,000528	0,000552	2
192	805	1000000	0,000805	0,001056	2
384	1588	1000000	0,001588	0,00161	2
768	5398	1000000	0,005398	0,003176	2
1536	11081	1000000	0,011081	0,010796	2
3072	23375	1000000	0,023375	0,022162	2
6144	49693	1000000	0,049693	0,04675	2
12288	10830	100000	0,1083	0,099386	2
24576	22705	100000	0,22705	0,2166	2



Mediciones de tiempo para la **tercera versión**:

n	*t_sum3 (ms)*	*repeticiones*	*t / repeticiones (ms)*	*t_calculado*	*constante*
3	104	1000000	0,000104		
6	164	1000000	0,000164	0,000208	2
12	308	1000000	0,000308	0,000328	2
24	333	1000000	0,000333	0,000616	2
48	539	1000000	0,000539	0,000666	2
96	828	1000000	0,000828	0,001078	2
192	1755	1000000	0,001755	0,001656	2
384	3066	1000000	0,003066	0,00351	2
768	11057	1000000	0,011057	0,006132	2
1536	16044	1000000	0,016044	0,022114	2
3072	30742	1000000	0,030742	0,032088	2
6144	49544	1000000	0,049544	0,061484	2
12288	11931	100000	0,11931	0,099088	2
24576	19482	100000	0,19482	0,23862	2
49152	43532	100000	0,43532	0,38964	2
98304	7871	10000	0,7871	0,87064	2



Para las tres versiones del algoritmo, la complejidad es lineal, es decir, O(n). Esto lo podemos confirmar porque, al calcular tiempos con la fórmula vista en teoría (utilizando O(n) como complejidad del método), y comparar estos con los tiempos obtenidos, obtenemos una constante de 2.

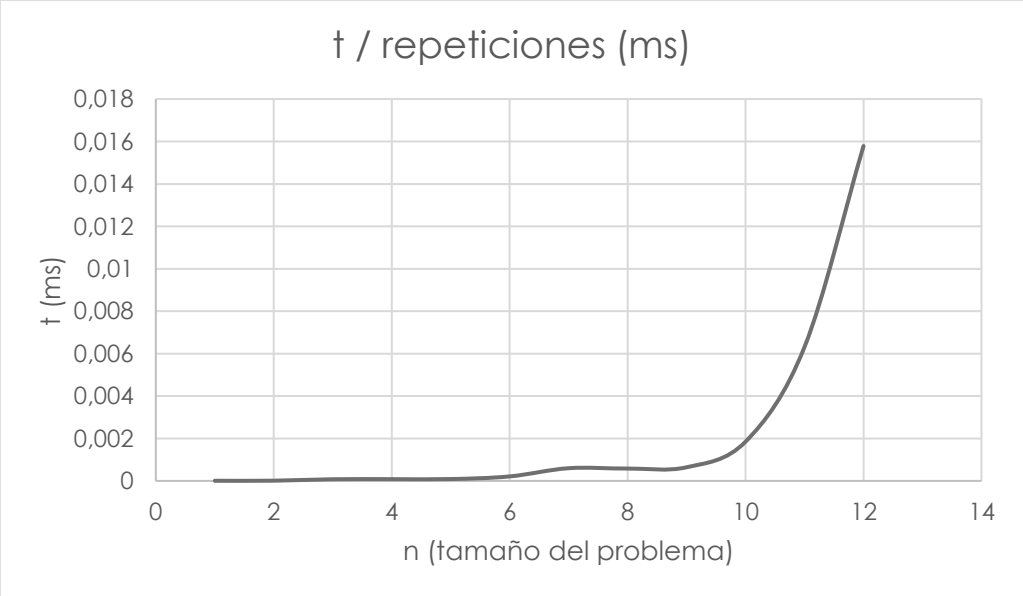
SUSTRACCION 4

Este algoritmo debe tener una complejidad $O(3^{n/2})$ mediante sustracción. Por ello, necesita tener 3 llamadas recursivas (a) y que en cada llamada se reste 2 (b). La complejidad del resto del método debe ser 0 (k).

```
public static boolean rec4(int n) {
    if (n <= 0)
        cont++;
    else {
        cont++;
        rec4(n - 2);
        rec4(n - 2);
        rec4(n - 2);
    }
    return true;
}
```

Mediciones:

n	*t_sustraccion4 (ms)*	*repeticiones*	*t / repeticiones (ms)*	*t_calculado*	*constante*
1	28	5000000	0,0000056		
2	51	5000000	0,0000102	9,69948E-06	1,732050808
3	387	5000000	0,0000774	1,76669E-05	1,732050808
4	412	5000000	0,0000824	0,000134061	1,732050808
5	431	5000000	0,0000862	0,000142721	1,732050808
6	1056	5000000	0,0002112	0,000149303	1,732050808
7	2991	5000000	0,0005982	0,000365809	1,732050808
8	2913	5000000	0,0005826	0,001036113	1,732050808
9	3214	5000000	0,0006428	0,001009093	1,732050808
10	9246	5000000	0,0018492	0,001113362	1,732050808
11	31577	5000000	0,0063154	0,003202908	1,732050808
12	78933	5000000	0,0157866	0,010938594	1,732050808



DIVISION 4

Este algoritmo debe tener una complejidad $O(n^2)$ mediante división. También debe tener 4 llamadas recursivas (a). Por ello, la complejidad del resto del método debe ser 2 (k) para que se cumpla la complejidad final. Debemos darle a b un valor tal que $a < b^k$. En mi caso, he elegido b = 3.

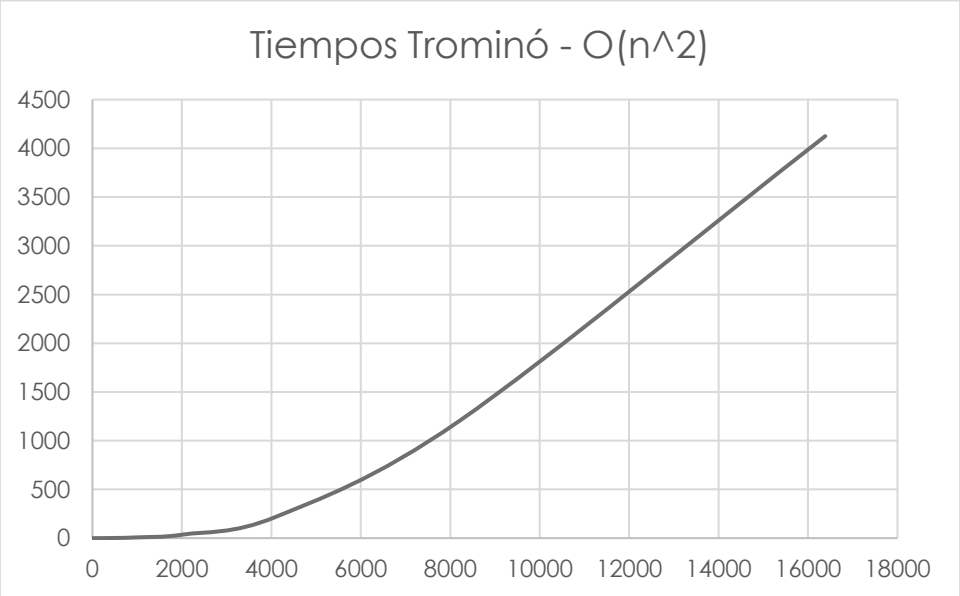
```java
public static boolean rec4(int n) {
    if (n <= 0)
        cont++;
    else {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                cont++;
            }
        }
        rec4(n / 3);
        rec4(n / 3);
        rec4(n / 3);
        rec4(n / 3);
    }
    return true;
}
```

n	*t_division4 (ms)*	*repeticiones*	*t / repeticiones (ms)*	*t_calculado*	*constante*
1	44	1000000	0,000044		
2	125	1000000	0,000125	0,000176	4
3	139	1000000	0,000139	0,00028125	2,25
4	163	1000000	0,000163	0,000247111	1,777777778
5	174	1000000	0,000174	0,000254688	1,5625
6	294	1000000	0,000294	0,00025056	1,44
7	521	1000000	0,000521	0,000400167	1,361111111
8	813	1000000	0,000813	0,00068049	1,306122449
9	1274	1000000	0,001274	0,001028953	1,265625
10	1619	1000000	0,001619	0,00157284	1,234567901
11	1961	1000000	0,001961	0,00195899	1,21
12	2293	1000000	0,002293	0,002333752	1,190082645

TROMINÓ

Mediciones de tiempo:

n	t_tromino (ms)	repeticiones	t / repeticiones (ms)	t_calculado	constante
2	87	27000	0,003222222		
4	90	27000	0,003333333	0,006444444	2
8	147	27000	0,005444444	0,006666667	2
16	300	27000	0,011111111	0,010888889	2
32	467	27000	0,017296296	0,022222222	2
64	1201	27000	0,044481481	0,034592593	2
128	3978	27000	0,147333333	0,088962963	2
256	14514	27000	0,537555556	0,294666667	2
512	57002	27000	2,111185185	1,075111111	2
1024	8390	1000	8,39	4,22237037	2
2048	38155	1000	38,155	16,78	2
4096	21782	100	217,82	76,31	2
8192	1199	1	1199	435,64	2
16384	4125	1	4125	2398	2



En mi implementación del algoritmo, la complejidad es $O(n^2)$. Esto se debe a que el método recursivo tiene 4 llamadas recursivas, en las que se reduce el parámetro por división entre 2. La complejidad del resto del método es O(1).

Al representar los datos en la gráfica, podemos ver cómo estos crecen de forma no lineal. Antes de confirmar la complejidad, calculo los tiempos que se deberían obtener mediante la fórmula vista en teoría. Al comparar estos tiempos con los obtenidos, obtenemos una constante de 2, por lo que podemos afirmar que el algoritmo tiene complejidad cuadrática.