Divide y vencerás v2 – Raúl Fernández España – UO278036

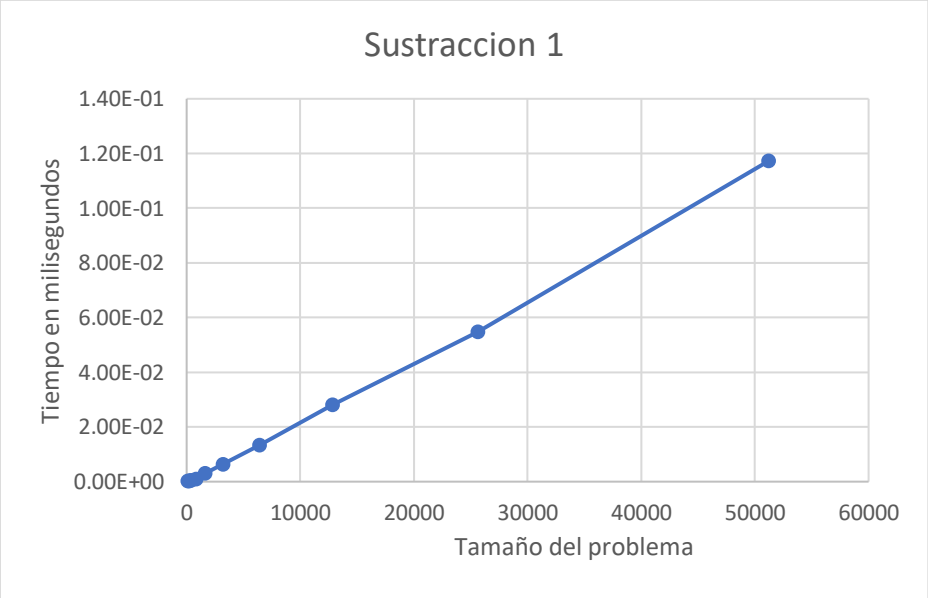
Las mediciones que se mostraran a continuación han sido realizadas en diferentes maquinas, pero siempre respetando el uso de una misma máquina para un ámbito de problema. Por lo tanto, se indica antes de cada análisis en que maquina ha sido realizado.

SUSTRACCIÓN:

Los siguientes datos han sido tomados de una maquina con las siguientes características, un procesador I7 8750H con 8Gb de memoria RAM.

Sustracción 1

N	tiempo	nVeces
100	1,44E-04	500000
200	2,26E-04	500000
400	4,24E-04	500000
800	8,58E-04	500000
1600	0,003008	500000
3200	0,006388	500000
6400	0,0132788	500000
12800	0,028078	500000
25600	0,054722	500000
51200	0,117242	500000

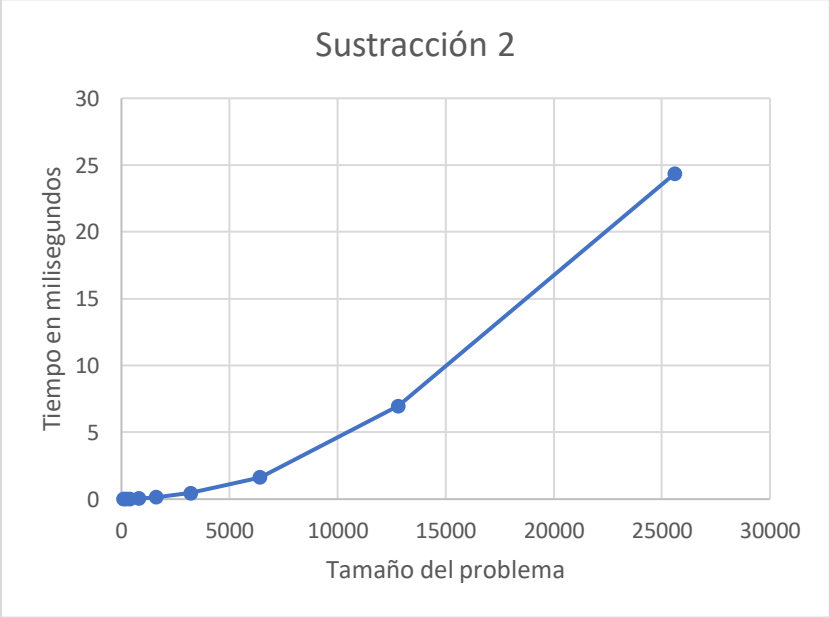


Para calcular la complejidad de este método recursivo podemos observar sus siguientes valores; a=1, b=1 y k=0. Podemos calcular rápidamente si complejidad observando que esta se trata de una complejidad O(n). Al observar la grafica podemos observar que esto se cumple, ya que también representa una complejidad lineal

Sustracción 2

Para este método contamos con los siguientes valores para calcular su complejidad a = 1, b = 1, k = 1. Por lo que la complejidad se puede deducir que es O(n^2), cuadrática.

N	tiempo	nVeces
100	0,001425	40000
200	0,003375	40000
400	0,01055	40000
800	0,0408	40000
1600	0,141925	40000
3200	0,46445	40000
6400	1,62375	40000
12800	6,979	1000
25600	24,367	1000
51200	x	x

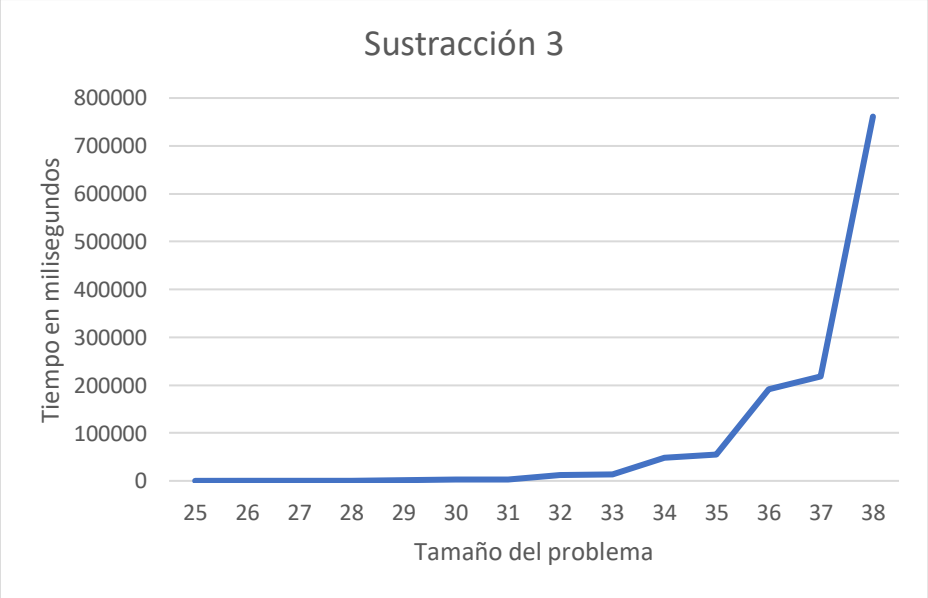


Como podemos observar en los datos y claramente en el grafico que representa la relación tamaño del problema/tiempo podemos observar que se cumple la complejidad cuadrática.

Sustracción 3

Para este método recursivo podemos observar que tenemos los siguientes valores a=2, b=1 y k=0. Por lo que teóricamente el resultado es $O(2^n)$.

N	tiempo	nVeces
24	47,21	100
25	52,99	100
26	190	1
27	213	1
28	761	1
29	874	1
30	3022	1
31	3409	1
32	12097	1
33	13805	1
34	48517	1
35	54826	1
36	191653	1
37	217915	1
38	761241	1



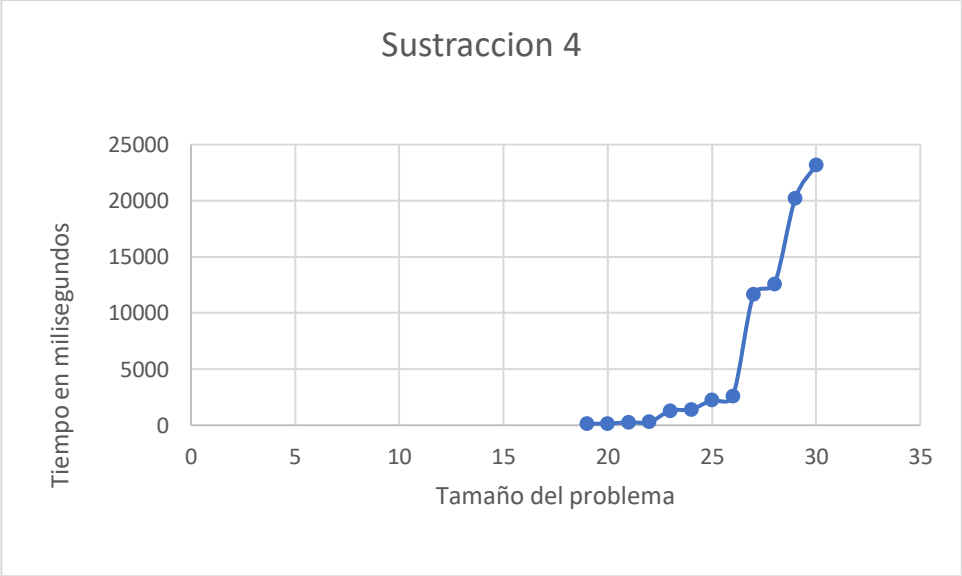
Como podemos observar en la tabla de valores para las n los tiempos se incrementan considerablemente rápido. Además, podemos observar como para valores superiores a 30 el tiempo va siendo extremadamente considerable aumentando cada vez mas rápido.

A partir de 38 valores el tiempo de espera supera la media hora, como podemos observar la complejidad teórica también se cumple en este caso

Sustracción 4

Este algoritmo ha sido construido para la siguiente complejidad $O(3^{n/2})$, a continuación, podemos observar los resultados experimentales.

N	tiempo	nVeces
19	144	1000
20	154	1000
21	263	1000
22	285	1000
23	1295	1000
24	1407	1000
25	2237	1000
26	2566	1000
27	11652	1000
28	12576	1000
29	20204	1000
30	23173	1000



Como se observa en el resultado a partir de n 19 hasta n 30 se puede observar como el coste temporal para el tamaño del problema aumenta de acorde con el marco teórico.

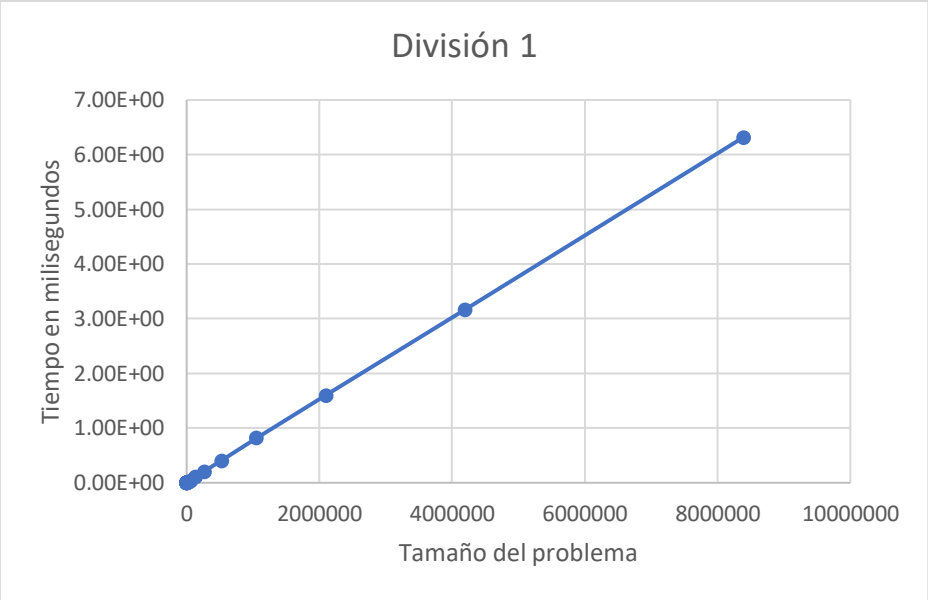
Por lo que podemos asumir que el algoritmo esta bien construido, sin embargo, se puede observar en sus tiempos que para mayor coste temporal los resultados experimentales se alejan mas de los resultados teóricos.

Los problemas de División han sido tomados bajo las condiciones de esta máquina, con procesador Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz 3.60 GHz y 8,00 GB de RAM.

División 1

Para este método recursivo contamos con los siguientes valores debido al algoritmo a=1, b=3 y k=1. Por lo que la complejidad es 0(n) a continuación los resultados experimentales para comparar

N	tiempo	nVeces
1	4,07E-08	1500000000
2	2,16E-06	1500000000
4	5,04E-06	1500000000
8	7,44E-06	1500000000
16	1,25E-05	1500000000
32	2,38E-05	1500000000
64	3,97E-05	1500000000
128	9,20E-05	1500000
256	1,75E-04	1500000
512	3,05E-04	1500000
1024	5,93E-04	1500000
2048	0,00120733	1500000
4096	0,00237067	1500000
8192	0,00465733	1500000
16384	0,00935133	1500000
32768	0,01862227	1500000
65536	0,037248	1500000
131072	0,104	1500
262144	0,20333333	1500
524288	0,4026666	1500
1048576	0,8173334	1500
2097152	1,598667	1500
4194304	3,165333	1500
8388608	6,316	1500

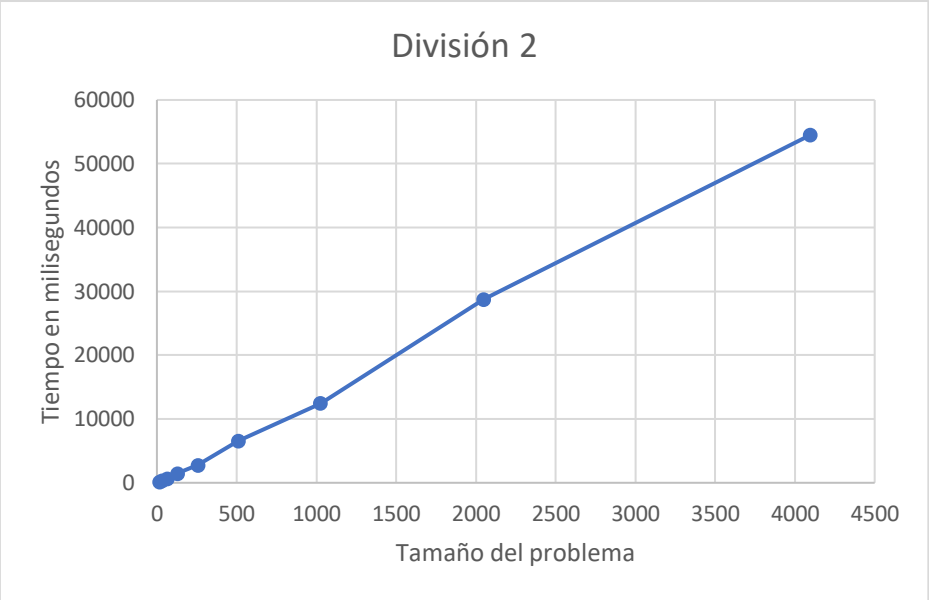


Como podemos observar gracias a la toma de datos y los resultados gráficos de la relación tiempo/tamaño del problema se puede observar que el algoritmo cumple el marco teórico y tiene una complejidad lineal O(n)

División 2

Para este método contamos con los valores del algoritmo a=2, b=2 y k=1. Por lo que según el marco teórico la complejidad del algoritmo es O(n log n) y una complejidad de pila de log n

N	tiempo	nVeces
16	114	1500000
32	304	1500000
64	584	1500000
128	1473	1500000
256	2775	1500000
512	6571	1500000
1024	12448	1500000
2048	28758	1500000
4096	54498	1500000



Como podemos observar en la gráfica también se cumple la complejidad O(n log n) por lo que se el resultado experimental se ajusta al marco teórico.

División 3

Para este método recursivo contamos con los siguientes valores a= 2, b=2 y k=0. Por lo que la complejidad lineal es 0(n) y la complejidad de pila es O(log n)

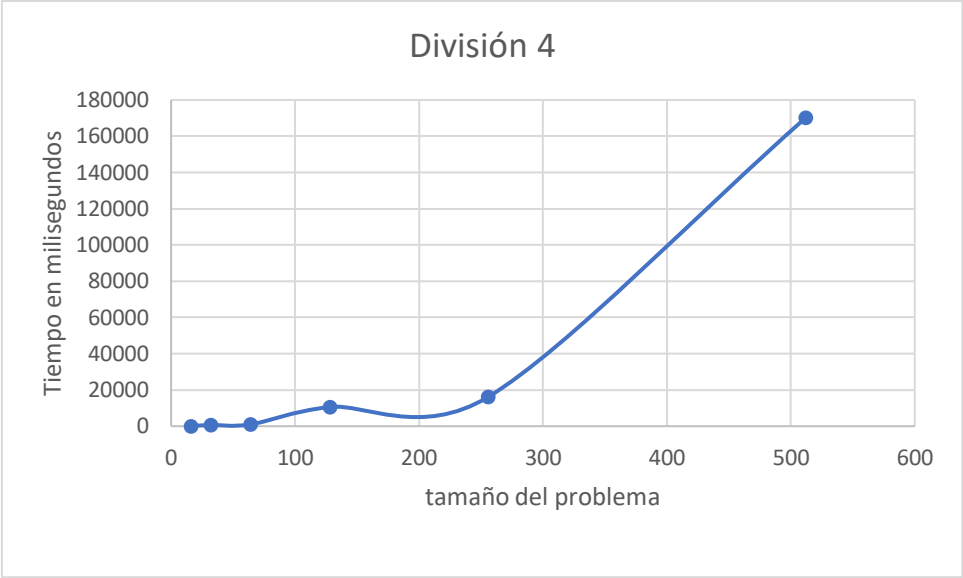
N	tiempo	nVeces
8	53	1500000
16	64	1500000
32	218	1500000
64	269	1500000
128	883	1500000
256	1062	1500000
512	3559	1500000
1024	4271	1500000
2048	14279	1500000
4096	17194	1500000
8192	56888	1500000
16384	68757	1500000

Como podemos observar con echarle un vistazo al resultado grafico de la relación tiempo tamaño del problema podemos observar que la complejidad es O(n). Por lo tanto, el algoritmo se ajusta al marco teórico.

División 4

Se a pedido un algoritmo con complejidad $O(n^2)$, cuyo algoritmo ha sido implementado, a continuación, las medidas experimentales para comprobar si el algoritmo tiene la complejidad pedida.

N	tiempo	nVeces
16	63	100000
32	658	100000
64	1006	100000
128	10620	100000
256	16218	100000
512	170160	100000

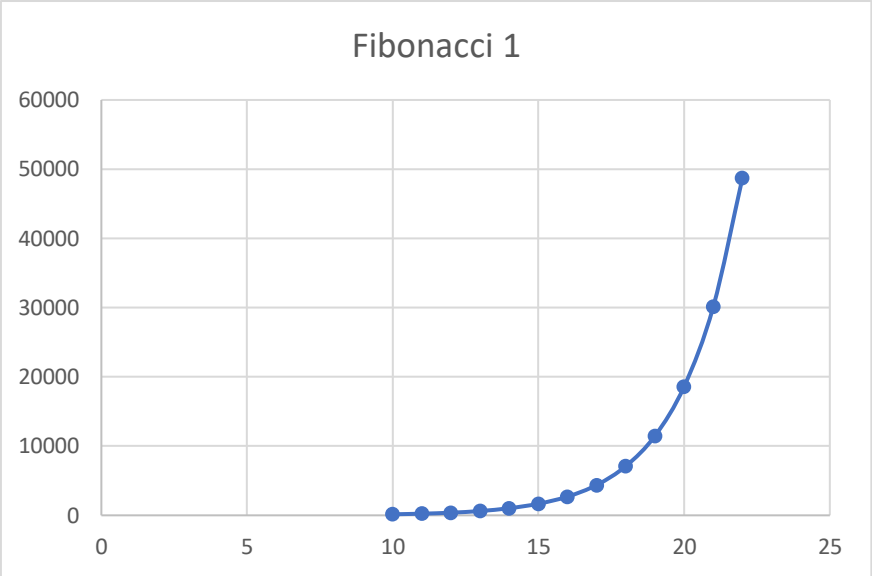


Como podemos ver el tiempo se aleja bastante del marco teórico, esto se debe a las optimizaciones internas de Java por lo tanto los resultados experimentales no se ajustan al marco teórico.

A continuación, vamos a analizar las diferencias entre los diferentes métodos de Fibonacci:

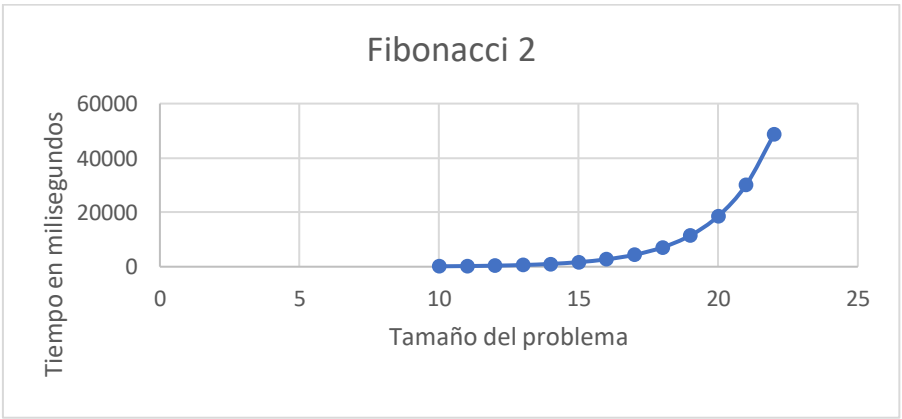
Método 1, resultados experimentales:

N	tiempo	nVeces
10	242	100000000
11	305	100000000
12	329	100000000
13	364	100000000
14	411	100000000
15	460	100000000
16	499	100000000
17	500	100000000
18	487	100000000
19	491	100000000
20	517	100000000
21	574	100000000
22	634	100000000



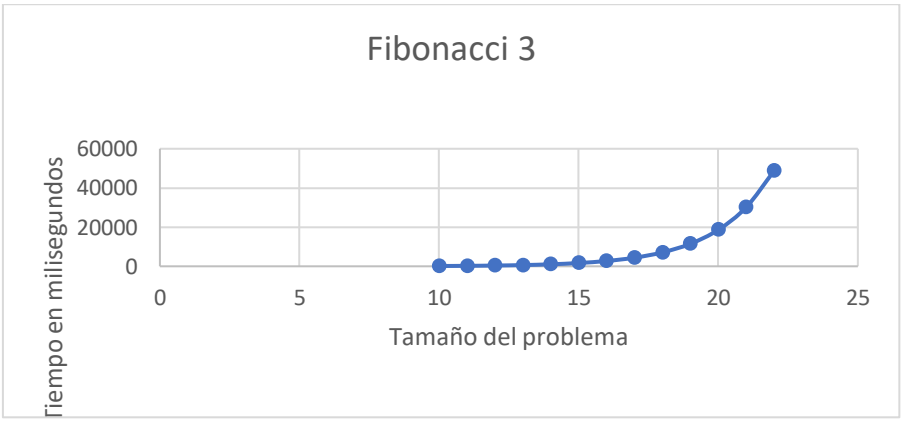
Método 2, resultados experimentales:

N	tiempo
10	367
11	445
12	484
13	536
14	528
15	565
16	613
17	681
18	666
19	727
20	770
21	836
22	825



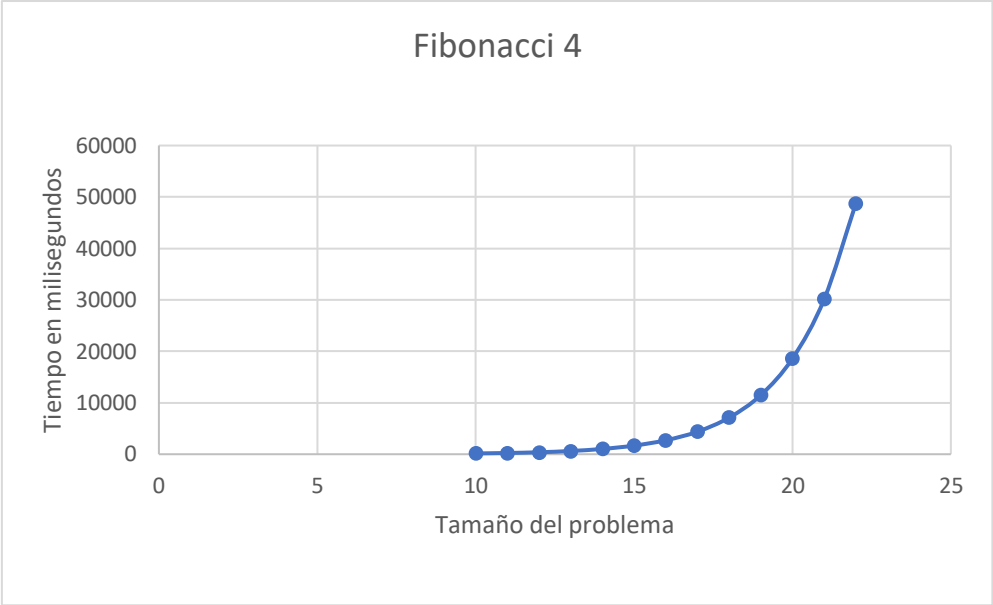
Método 3, resultados experimentales

N	tiempo	nVeces
10	702	100000000
11	672	100000000
12	776	100000000
13	781	100000000
14	886	100000000
15	902	100000000
16	1002	100000000
17	1030	100000000
18	1177	100000000
19	1168	100000000
20	1370	100000000
21	1392	100000000
22	1530	100000000



Método 4, resultados experimentales

N	tiempo	nVeces
10	151	1000000
11	240	1000000
12	383	1000000
13	626	1000000
14	1030	1000000
15	1676	1000000
16	2711	1000000
17	4384	1000000
18	7113	1000000
19	11484	1000000
20	18607	1000000
21	30128	1000000
22	48726	1000000



Como podemos observar los tres primeros métodos comparten complejidad, sin embargo, para diferentes tamaños del problema podemos observar que existen diferencias de tiempos entre ellos. El más rápido el primero siguiéndolo el segundo y por ultimo el tercero, por tanto, a pesar de compartir complejidad, el coste temporal varia de unos a otros.

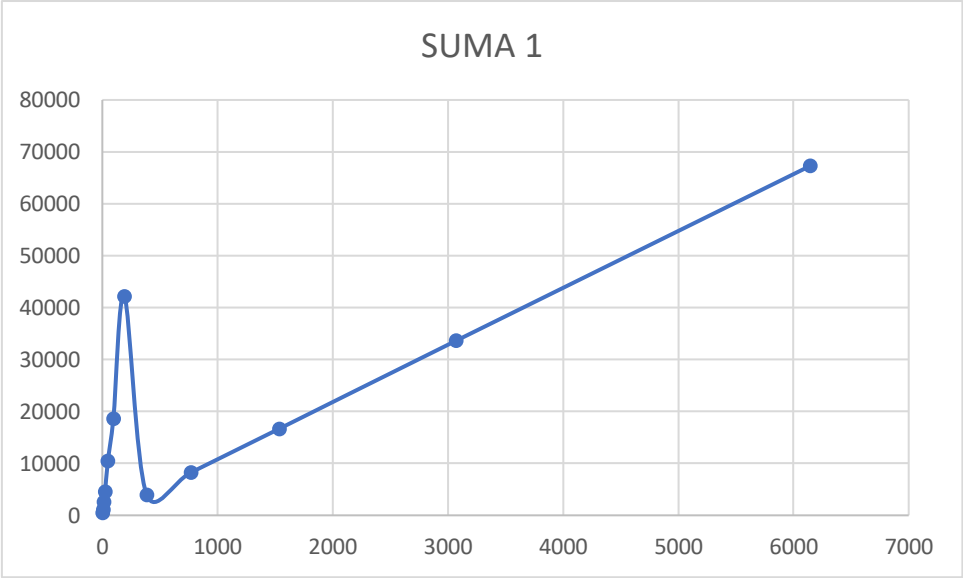
En el segundo al usar la complejidad temporal aumenta respecto al primero(iterativo), y el tercero, que utiliza recursividad aumenta su coste temporal respecto al segundo.

Por último el método 4 es el único que posee otra complejidad, $O(1.6^n)$, se puede observar claramente en la grafica que tiene una complejidad exponencial cada vez que la n aumenta en uno el tiempo se ve multiplicado por 1,6.

A continuación vamos a analizar el vectorSum 1:

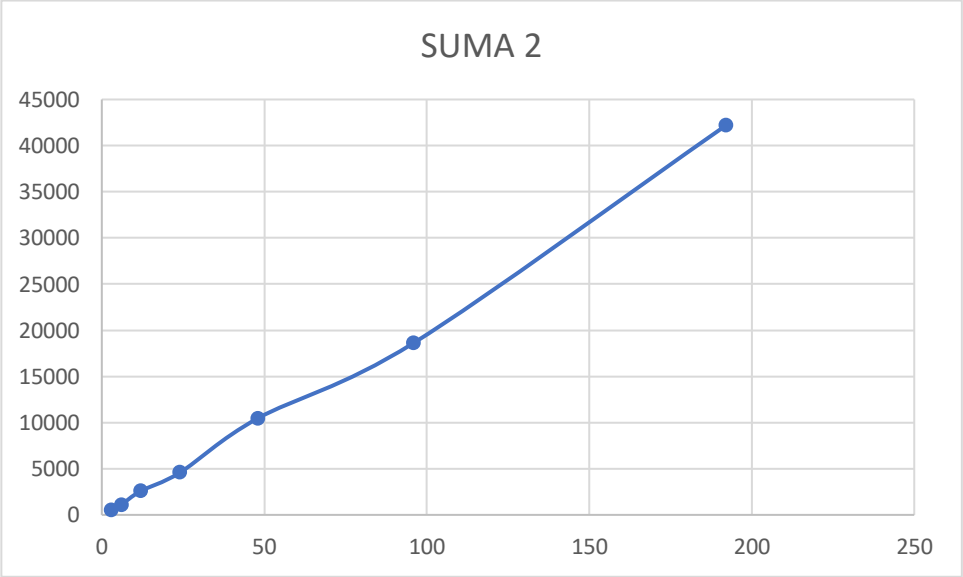
Método 1:

N	tiempo	nVeces
3	177	100000000
6	241	100000000
12	301	100000000
24	504	100000000
48	941	100000000
96	2103	100000000
192	2711	100000000
384	4012	100000000
768	8244	100000000
1536	16680	100000000
3072	33641	100000000
6144	67286	100000000



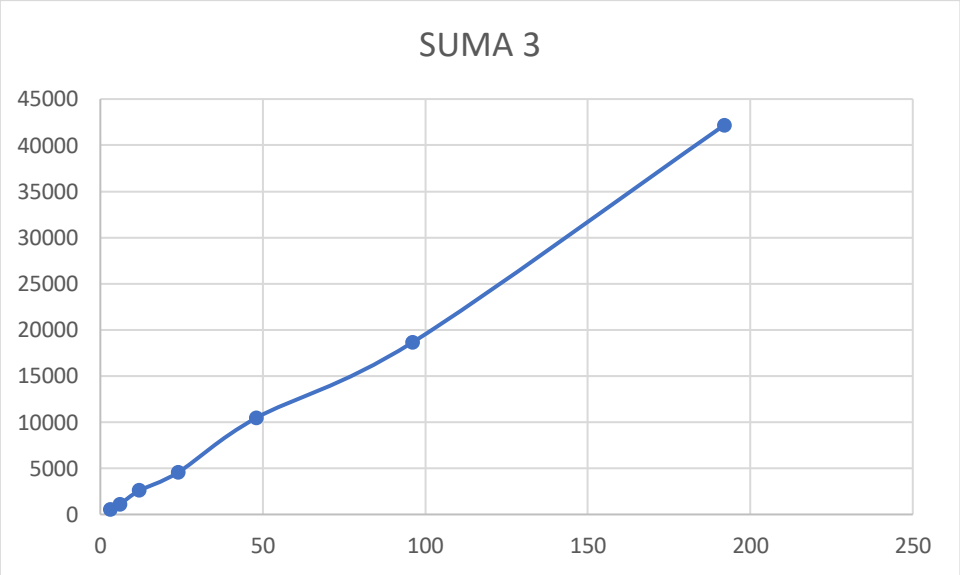
Método 2:

N	tiempo
3	271
6	544
12	1021
24	2017
48	7412
96	22087
192	47035



Método 3:

N	tiempo
3	499
6	1098
12	2587
24	4575
48	10462
96	18631
192	42174



Como se puede observar la complejidad de los 3 algoritmos de suma es lineal, O(n), sin embargo, se pueden observar significables diferencias de coste temporal.

Como podemos ver el coste temporal del primero es el menos, siguiéndolo el segundo y por último el tercero.

Esto se debe a como están construidos los algoritmos, el numero 1 calcula iterativamente la suma de un vector, el método 2 lo calcula recursivamente, lo que como podemos observar aumenta el coste temporal respecto al anterior.

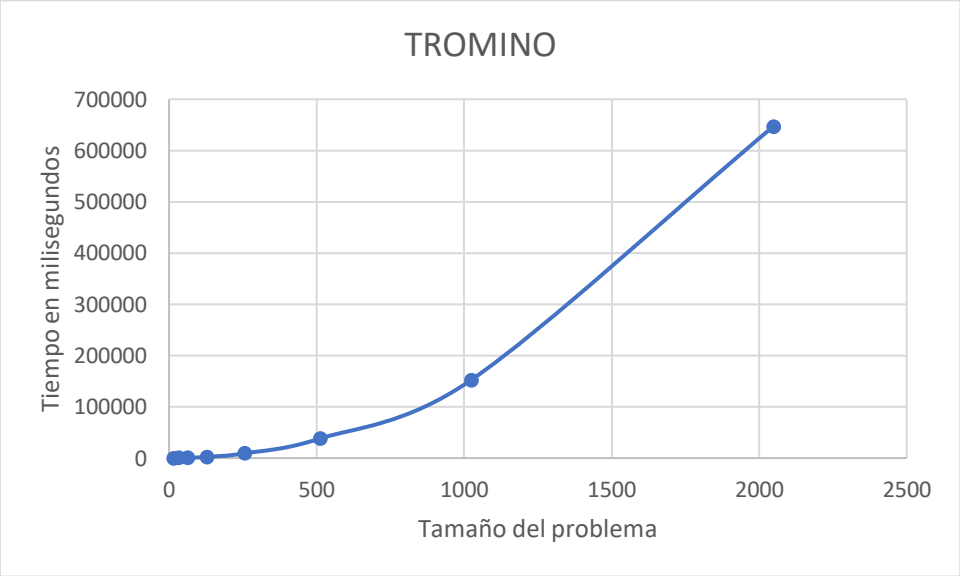
Por último el método 3 también calcula los tiempos haciendo la suma iterativa de un vector, sin embargo, tiene mas coste temporal debido a que este tiene dos llamadas recursivas mientras que el método 2 solo tiene una.

TROMINO

Una vez resulto y probado el algoritmo para resolver el tromino, se toman las siguientes medidas para comprobar la complejidad de el algoritmo junto a su coste temporal experimentalmente.

N	tiempo	nVeces
16	41	100000
32	145	100000
64	572	100000
128	2293	100000
256	9236	100000
512	38213	100000
1024	152301	100000
2048	646969	100000

Grafica del tromino:



Como podemos ver rápidamente observando su relación de tamaño del problema / tiempo podemos deducir que la complejidad del algoritmo es $O(n^2)$, cuadrática.

Por lo tanto esta complejidad cuadra con el marco teórico, teóricamente su complejidad es cuadrática debido a que se realizan cuatro llamadas recursivas (a=4) en cada iteración se divide el tamaño del cuadrante en 2 (b=2) y no hay ningún bucle extra por tanto (k=0).

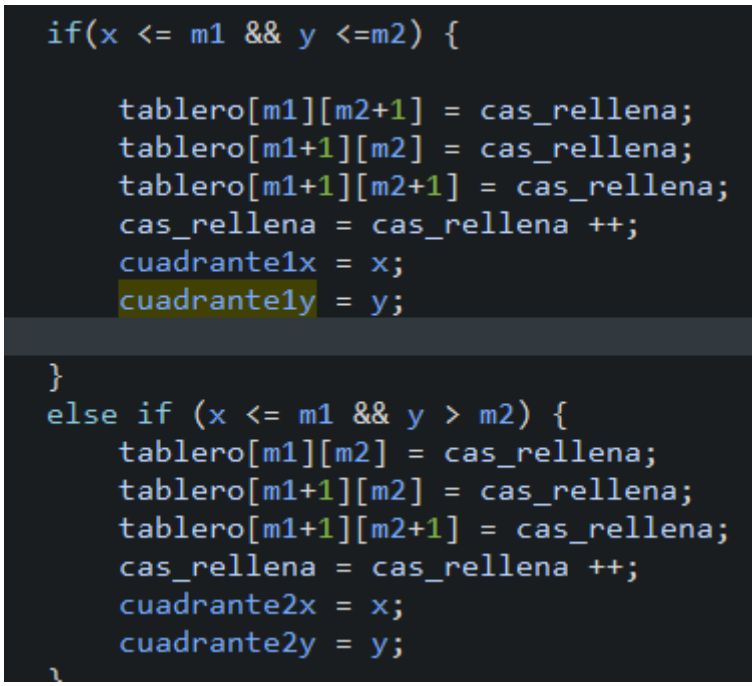
De esta forma calculamos $a>b^k$ por tanto si despejamos e resultado de la complejidad teórica seria $O(n^2)$

Lo cual se puede concluir que se corresponde con los resultados experimentales.

Las medidas han sido realizadas en un I7-10700K con 64Gb de RAM

Problema resuelto:

El fallo estaba en la siguiente sección de código



```
if(x <= m1 && y <=m2) {

    tablero[m1][m2+1] = cas_rellena;
    tablero[m1+1][m2] = cas_rellena;
    tablero[m1+1][m2+1] = cas_rellena;
    cas_rellena = cas_rellena ++;
    cuadrante1x = x;
    cuadrante1y = y;

}
else if (x <= m1 && y > m2) {
    tablero[m1][m2] = cas_rellena;
    tablero[m1+1][m2] = cas_rellena;
    tablero[m1+1][m2+1] = cas_rellena;
    cas_rellena = cas_rellena ++;
    cuadrante2x = x;
    cuadrante2y = y;
```

Al refactorizar el código al igualar los cuadrantes estaba ignorando la sección y igualando ambas con el cuadrante primero eliminando así el hueco -1 y rellenando de forma errónea todo el tablero dejando al final varios vacíos.

Modificando ese pequeño olvido ya rellena perfectamente el tablero, el tiempo no varía ya que había sido tomado antes de refactorizar el código, estando este correcto.