

# **PREGUNTAS SECCIÓN 3**

## **Sesiones prácticas 5-6**

Jonathan Arias Busto




UO283586

71780982-Y

Escuela de Ingeniería Informática - EII

## INFORMACIÓN SOBRE ORDENADOR USADO PARA MEDIR TIEMPOS

Procesador	Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz 2.50 GHz
RAM instalada	16,0 GB (15,8 GB usable)

- ▼  Adaptadores de pantalla
  -  Intel(R) UHD Graphics
  -  NVIDIA GeForce GTX 1650

Se trata de un portátil con el modo de energía en “High performance”.

Cambiar la configuración del plan: High performance

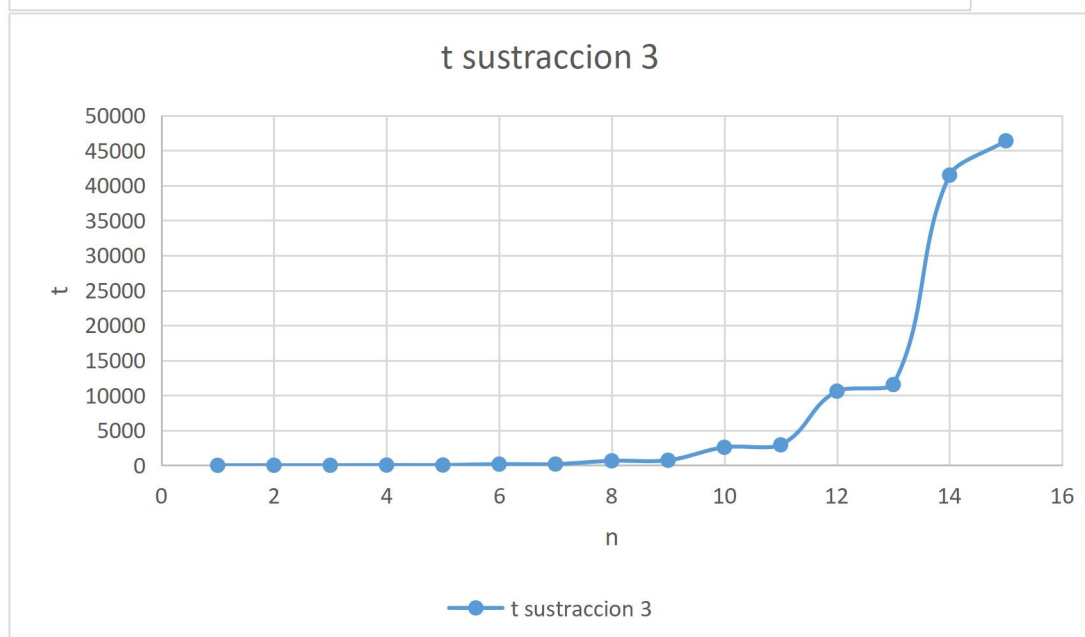
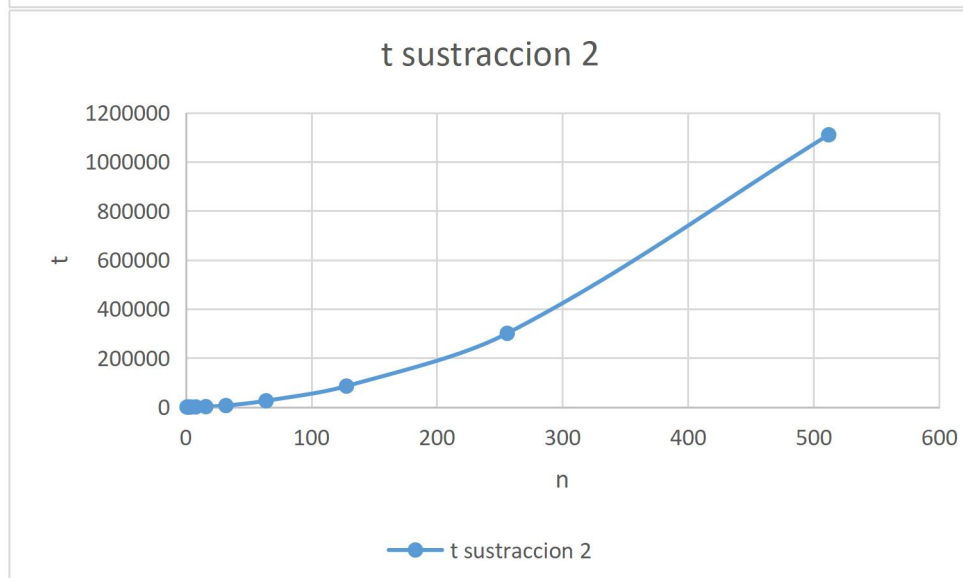
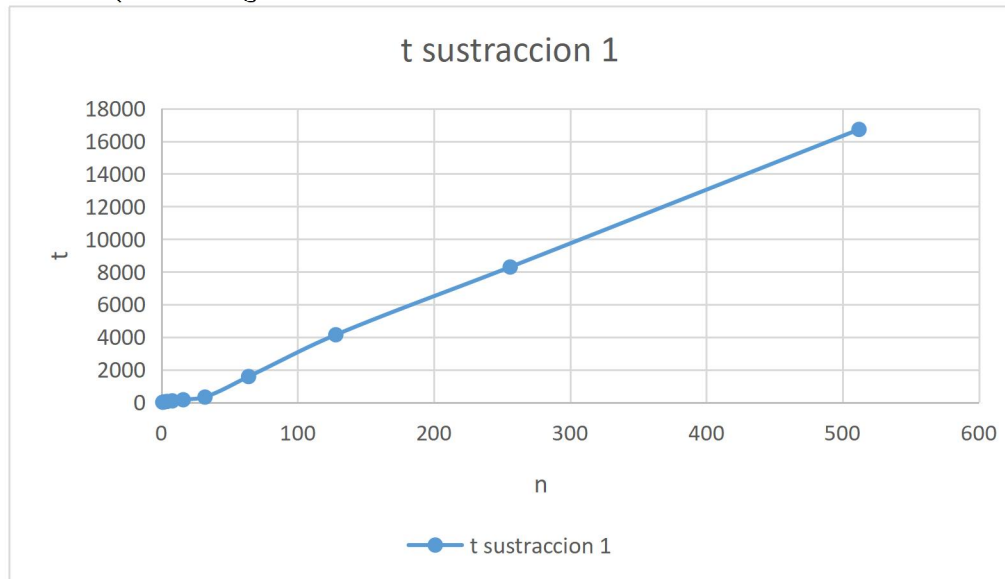
## TIEMPOS SUSTRACCIÓN

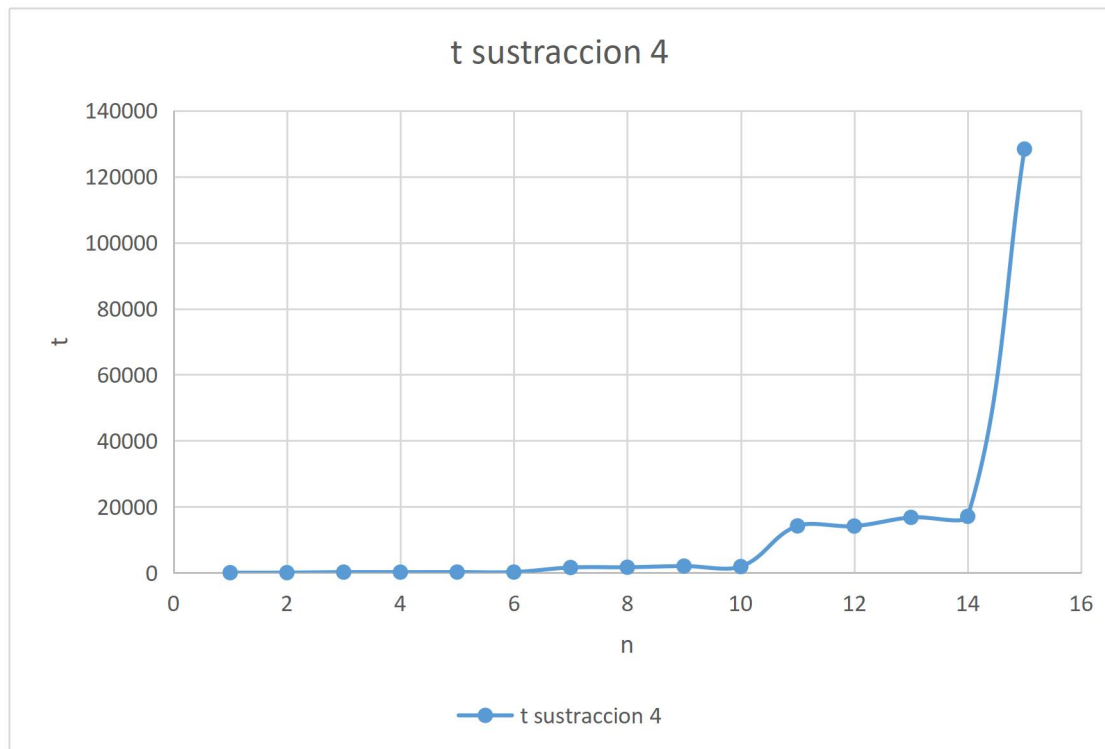
Los tiempos obtenidos después de ejecutar las diferentes versiones de Sustracción(1, 2, 3 y 4) son los siguientes:

n	t sustraccion 1	t sustraccion 2
1	10	44
2	31	95
4	56	195
8	93	520
16	160	1545
32	325	6151
64	1583	25602
128	4142	85472
256	8293	300637
512	16722	1110151
n veces =10000000		n veces = 10000000

n	t sustraccion 3	t sustraccion 4
1	5	8
2	14	20
3	9	173
4	40	168
5	42	192
6	169	192
7	180	1571
8	645	1656
9	727	2006
10	2580	1860
11	2930	14181
12	10603	14131
13	11545	16762
14	41477	17078
15	46382	128366
n veces = 1000000		n veces = 10000000

Las correspondientes gráficas son:





Podemos dividir la ejecución de los 4 diferentes algoritmos en 2 grupos:

- Sustracción 1 y 2:  $O(n)$  y  $O(n^2)$  respectivamente
- Sustracción 3 y 4:  $O(n^2)$  y  $O(3^{(n/2)})$  respectivamente

El grupo 1 tiene complejidades más bajas que las del grupo 2. Para comprobar que obtuvimos una complejidad correcta mirando el código podemos usar los tiempos obtenidos.

--> Sustracción 1:

$t_1 = 160$  para  $n_1 = 16$

$t_2 = X$  para  $n_2 = 32$

$K = 32/16 = 2$

$t_2 = K^{t_1} * t_1 = 2 * 160 = 320$

Se puede comprobar que el tiempo es más o menos correcto.

--> Sustracción 2:

$t_1 = 1545$  para  $n_1 = 16$

$t_2 = X$  para  $n_2 = 32$

$K = 32/16 = 2$

$t_2 = K^{t_1} * t_1 = 2^{1545} * 1545 = 6180$

Se puede comprobar que el tiempo es más o menos correcto.

--> Sustracción 3:

$t_1 = 2930$  para  $n_1 = 11$

$t_2 = X$  para  $n_2 = 12$

$t_2 = f(n_2) * t_1 / f(n_1) = 2^{12} * 2930 / 2^{11} = 5860$

Se puede comprobar que el tiempo medido no es el esperado, pero esto se debe a que la medición va por rangos de valores y crece mucho de repente.

--> Sustracción 4:

$t_1 = 1860$  para  $n_1 = 10$

$t_2 = X$  para  $n_2 = 11$

$K = 32/16 = 2$

$t_2 = t_2 = f(n_2) * t_1 / f(n_1) = 3^{11/2} * 1860 / 3^{10/2} = 3222$

Se puede comprobar que el tiempo medido no es el esperado, pero esto se debe a que la medición va por rangos de valores y crece mucho de repente.

Ahora podemos usar la formula de divide y vencerás de sustracción:

--> Sustracción 1:

$$a = 1$$

$$b = 1$$

$$k = 0$$

Por tanto  $O(n)$

--> Sustracción 2:

$$a = 1$$

$$b = 1$$

$$k = 1$$

Por tanto  $O(n^2)$

--> Sustracción 3:

$$a = 2$$

$$b = 1$$

$$k = 0$$

Por tanto  $O(2^n)$

--> Sustracción 4:

$$a = 3$$

$$b = 2$$

$$k = 0$$

Por tanto  $O(3^{(n/2)})$

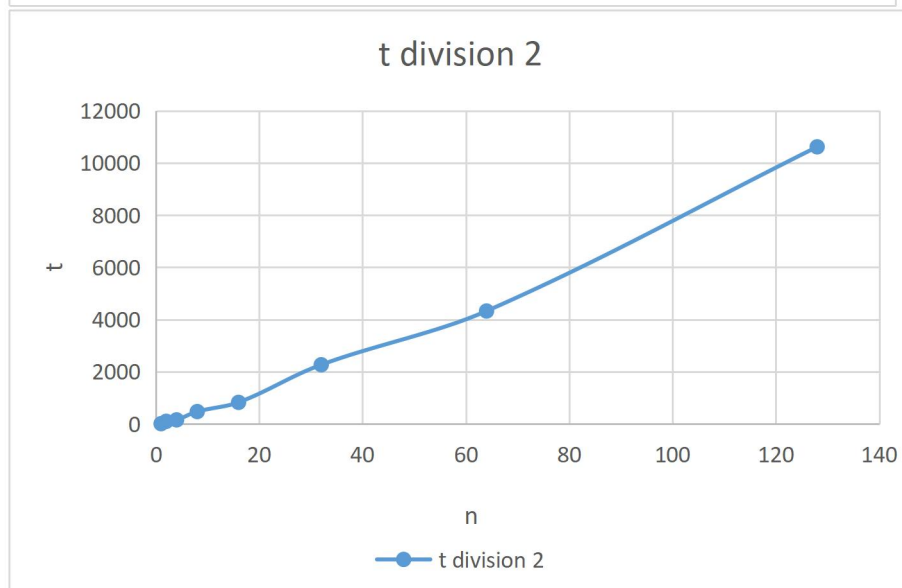
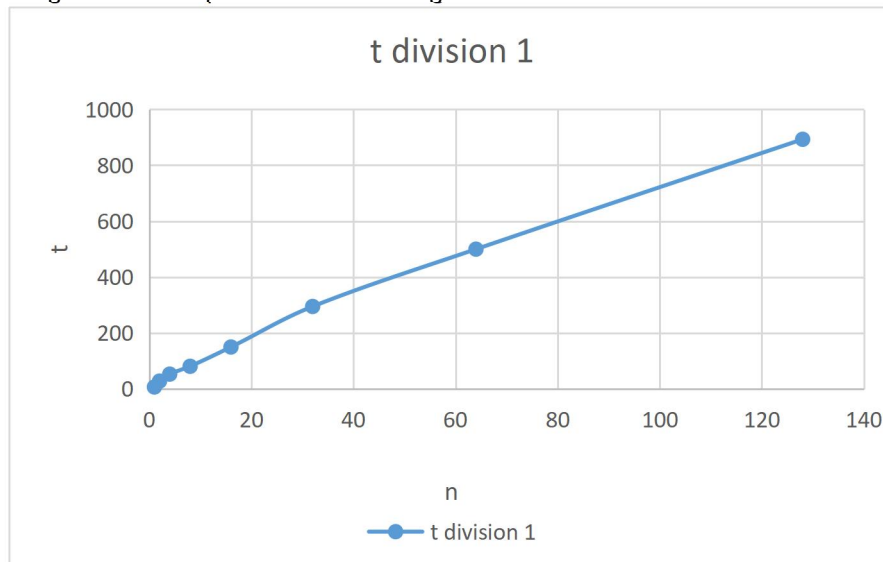
Podemos corroborar que sustracción 1 y 2 son esas complejidades, pero sustracción 3 y 4 al ser exponenciales no se puede comprobar ya que los tiempos medidos son confusos (o están por rangos).

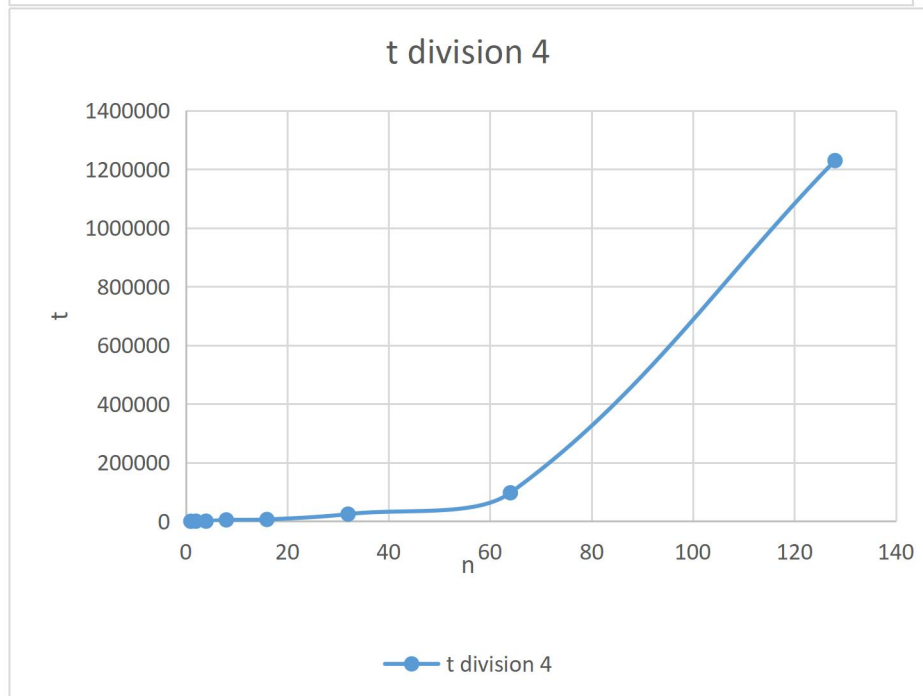
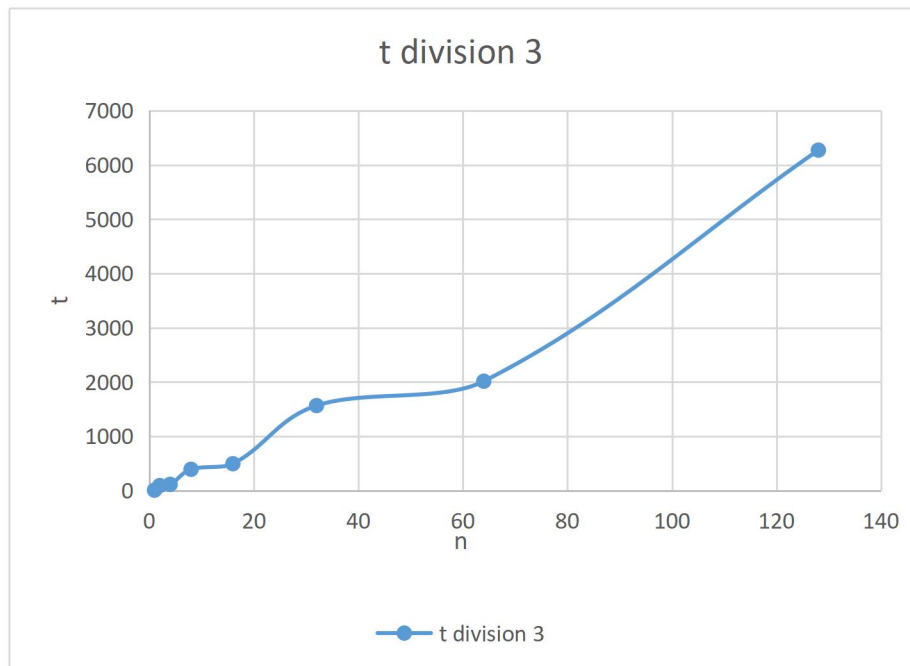
## TIEMPOS DIVISIÓN

Los tiempos obtenidos después de ejecutar las diferentes versiones de División(1, 2, 3 y 4) son los siguientes:

n	t division 1	t division 2	t division 3	t division 4
1	7	7	6	10
2	28	97	92	288
4	53	152	114	366
8	81	470	391	4636
16	150	823	494	6089
32	295	2265	1565	24457
64	500	4325	2014	97092
128	893	10619	6271	1229667
n veces = 10000000    n veces = 10000000    n veces = 10000000    n veces = 10000000				

Las gráficas correspondientes son las siguientes:





En este caso las complejidades mirando el código (esperadas) son:

- División 1:  $O(n)$
- División 2:  $O(n \log n)$
- División 3:  $O(n)$
- División 4:  $O(n^2)$



Como en el anterior caso vamos a comprobar a través de los tiempos si la complejidad se acerca a la realidad:

--> División 1:

$t_1 = 150$  para  $n_1 = 16$

$t_2 = X$  para  $n_2 = 32$

$K = 32/16 = 2$

$t_2 = K^{t_1} * t_1 = 2 * 150 = 300$

Se puede comprobar que el tiempo es más o menos correcto.

--> División 2:

$t_1 = 823$  para  $n_1 = 16$

$t_2 = X$  para  $n_2 = 32$

$K = 32/16 = 2$

$t_2 = f(n_2) * t_1 / f(n_1) = (32 * \log(32) * 823) / (16 * \log(16)) = 2057$

Se puede comprobar que el tiempo es más o menos correcto.

--> División 3:

$t_1 = 494$  para  $n_1 = 16$

$t_2 = X$  para  $n_2 = 32$

$K = 32/16 = 2$

$t_2 = K^{t_1} * t_1 = 2 * 494 = 988$

Se puede comprobar que el tiempo es más o menos correcto (aunque al no ser una  $O(n)$  pura ya que tiene un log en el exponente el tiempo sera superior a este).

--> División 4:

$t_1 = 6089$  para  $n_1 = 16$

$t_2 = X$  para  $n_2 = 32$

$K = 32/16 = 2$

$t_2 = K^{t_1} * t_1 = 2^{6089} * 6089 = 24356$

Se puede comprobar que el tiempo es más o menos correcto.

Usando las fórmulas teóricas de divide y vencerás para la división:

--> División 1:

$a = 1$

$b = 3$

$k = 1$

Por tanto  $O(n)$

--> División 2:

$a = 2$

$b = 2$

$k = 1$

Por tanto  $O(n \log n)$

--> División 3:

$a = 2$

$b = 2$

$k = 0$

Por tanto  $O(n)$

--> División 4:

$a = 4$

$b = 2$

$k = 0$

Por tanto  $O(n^2)$

# TIEMPOS FIBONACCI

Los tiempos obtenidos después de ejecutar las diferentes versiones de Fibonacci(1, 2, 3 y 4) son los siguientes:

n	t fibonacci 1	t fibonacci 2	t fibonacci 3	t fibonacci 4
10	37	47	90	5
11	42	66	94	9
12	53	64	99	10
13	53	61	96	24
14	50	71	111	24
15	49	76	111	33
16	52	76	125	49
17	62	82	130	71
18	61	88	140	112
19	60	90	140	161
20	65	92	156	224
21	71	94	169	334
22	69	100	185	547
23	65	111	194	881
24	71	114	212	1426
25	85	120	222	2318
26	88	127	223	3749

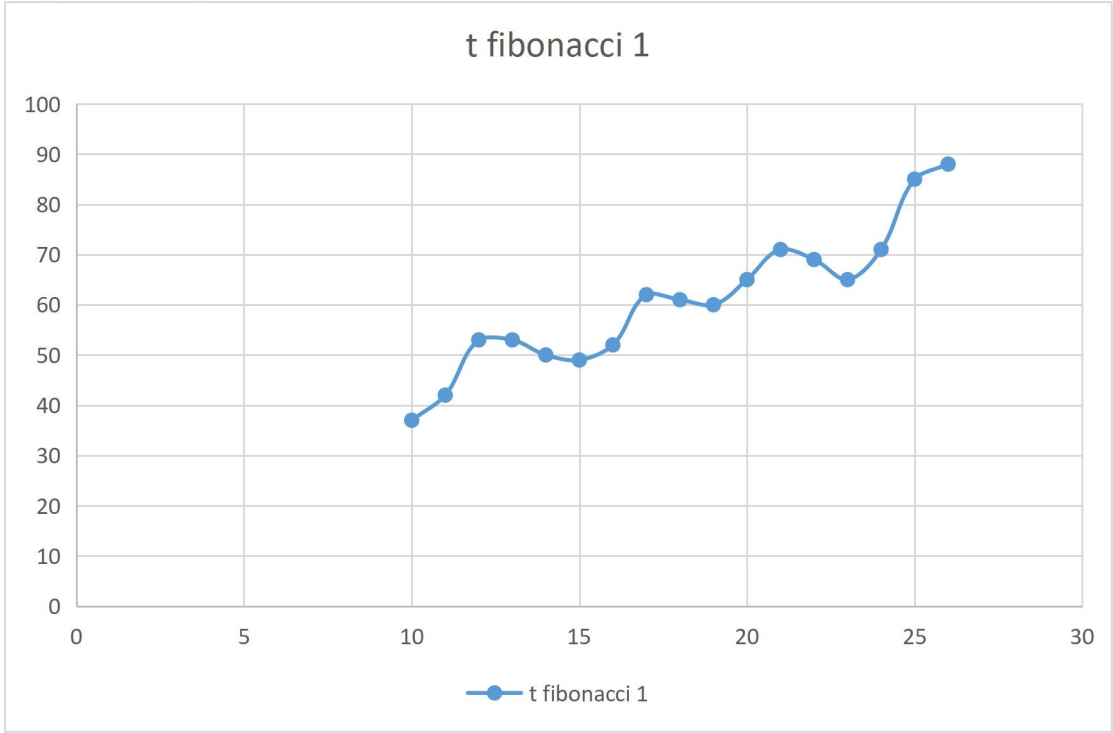
n veces = 10000000

n veces = 10000000

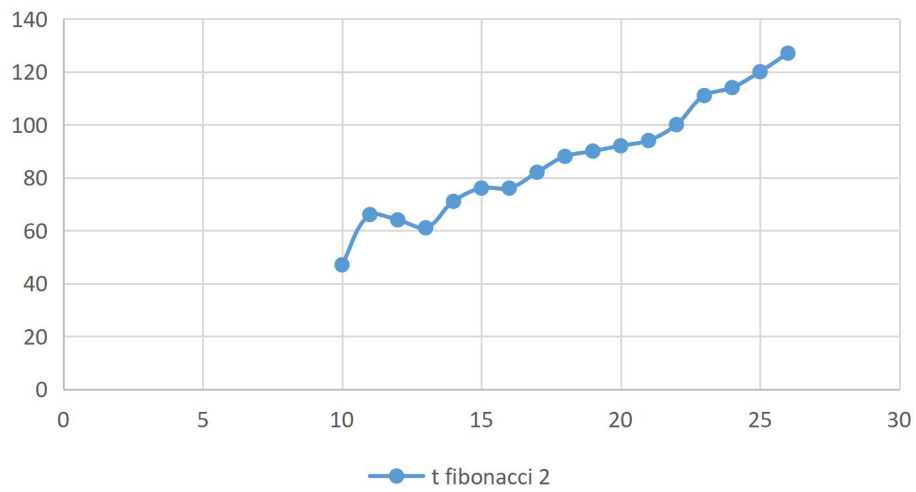
n veces = 10000000

n veces = 10000

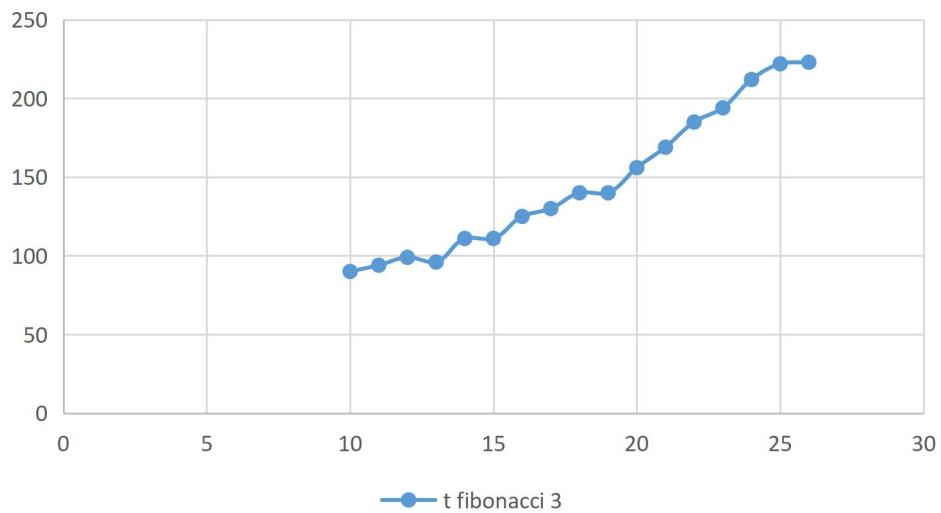
Las gráficas correspondientes son:



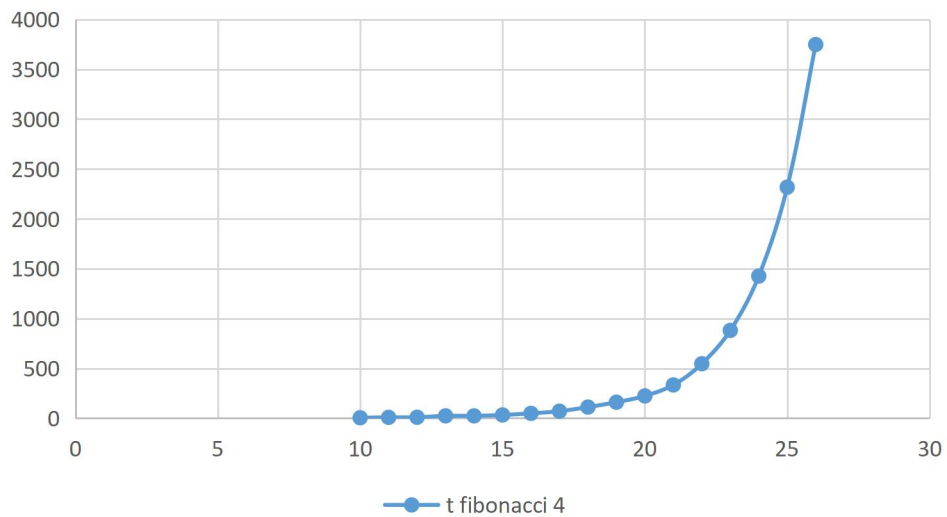
t fibonacci 2



t fibonacci 3



t fibonacci 4



En este caso las complejidades mirando el código (esperadas) son:

- Fibonacci 1:  $O(n)$
- Fibonacci 2:  $O(n)$
- Fibonacci 3:  $O(n^2)$
- Fibonacci 4: entre  $O(2^n)$  y  $O(2^{n/2})$  ya que no se puede calcular

Como se usan cargas tan pequeñas de trabajo no hay forma de calcular si el resultado es el esperado, ya que la diferencia de calcular el número fibonacci(30) y fibonacci(31) es muy pequeña, de ahí que no se pueda calcular con una fórmula matemática.

Lo que podemos usar para los casos recursivos son las fórmulas de divide y vencerás:

--> Fibonacci 3:

a = 1

b = 1

k = 0

Por tanto con la fórmula de dyv para la forma de sustracción tenemos  $O(n^2)$

--> Fibonacci 4:

Este caso es más complejo ya que tenemos que determinar una zona de trabajo del algoritmo. Siendo esta zona:  $O(2^{n/2}) \leq O(\text{fibonacci } 4) \leq O(2^n)$

Puesto que tenemos:

a = 2

b1 = 1 y b2 = 2 (De aquí que no se pueda determinar la complejidad)

k = 0

# TIEMPOS SUMAVECTOR

Los tiempos obtenidos después de ejecutar las diferentes versiones de SumaVector(1, 2 y 3) son los siguientes:

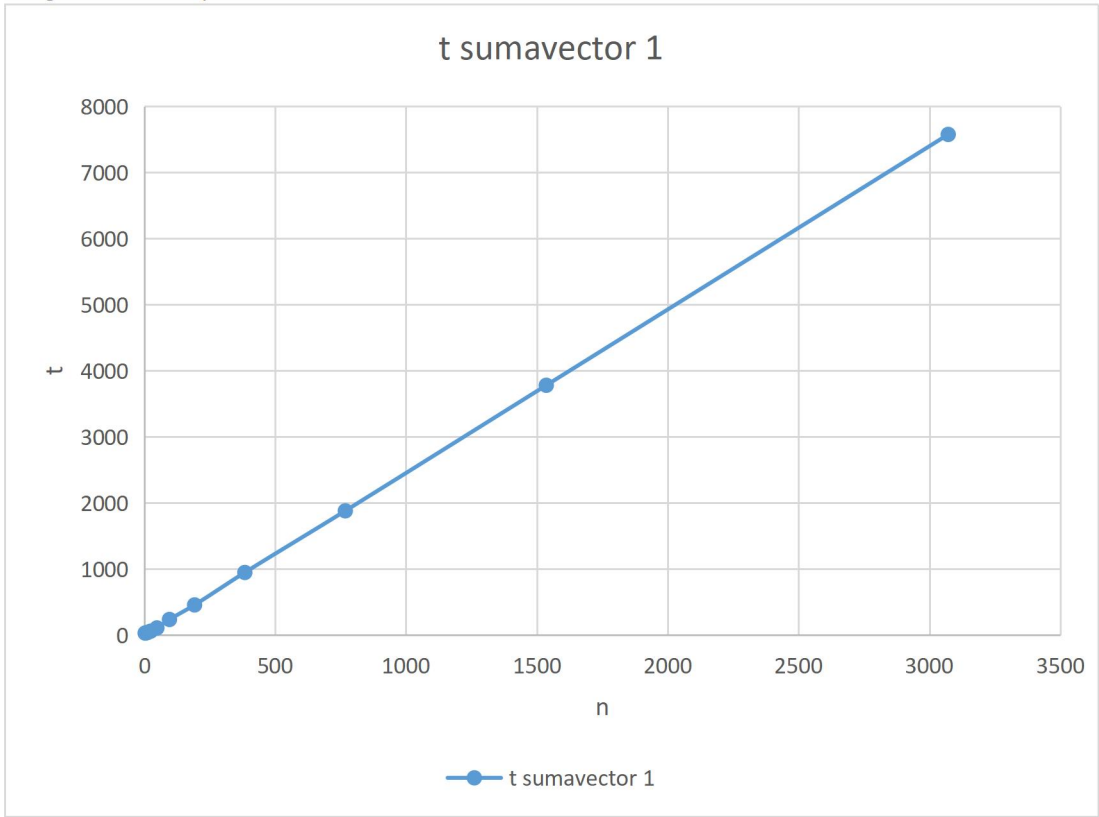
n	t sumavector 1	t sumavector 2	t sumavector 3
3	31	42	72
6	34	76	139
12	40	127	302
24	60	269	536
48	109	1013	1199
96	237	2973	2168
192	457	5923	4970
384	948	12738	9257
768	1881	37189	21128
1536	3781	75538	37106
3072	7577	154707	87599

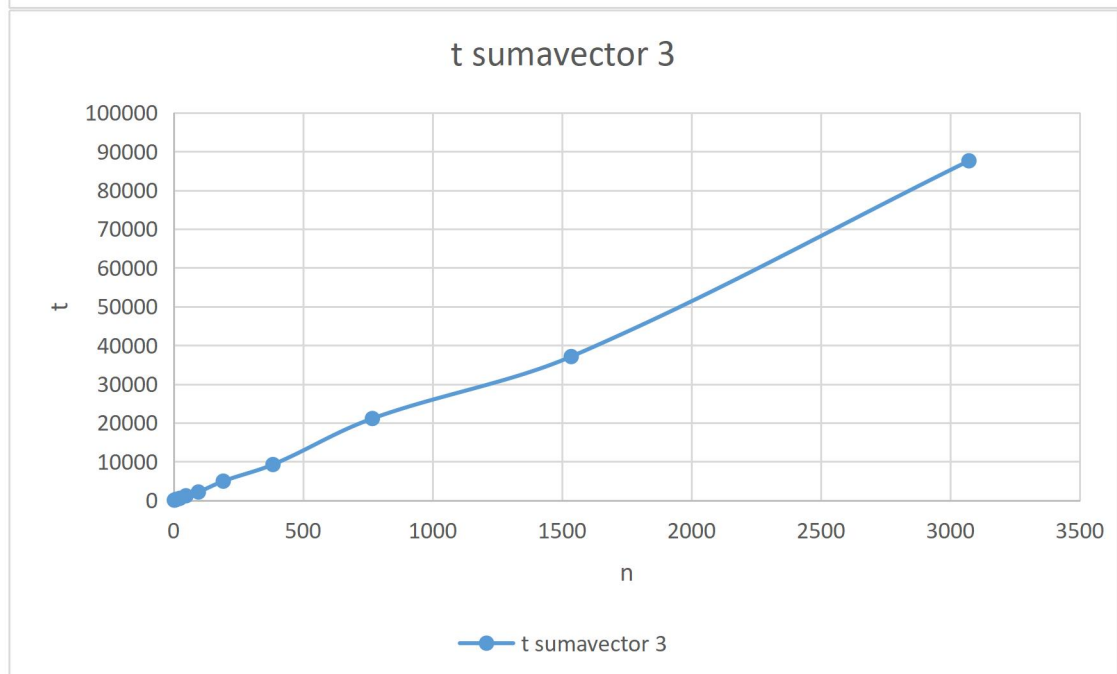
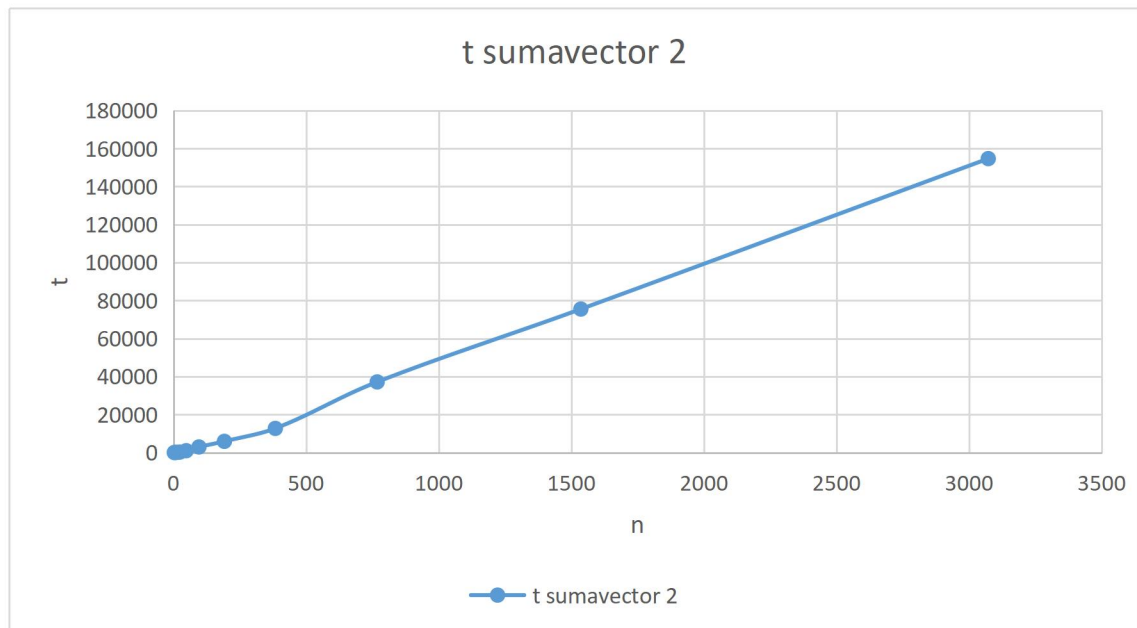
n veces = 10000000

n veces = 10000000

n veces = 10000000

Las gráficas correspondientes son:





Como en los anteriores casos predecimos las complejidades (esperadas) mirando el código:

- SumaVector1:  $O(n)$
- SumaVector2:  $O(n)$
- SumaVector3:  $O(n)$

Haciendo uso de los tiempos medidos de la ejecución:

--> SumaVector1:

$t_1 = 109$  para  $n_1 = 48$

$t_2 = X$  para  $n_2 = 96$

$K = 96/48 = 2$

$t_2 = K^{*1} * t_1 = 2 * 109 = 218$

Se puede comprobar que el tiempo es más o menos correcto.

--> SumaVector2:

$t_1 = 1013$  para  $n_1 = 48$

$t_2 = X$  para  $n_2 = 96$

$K = 96/48 = 2$

$t_2 = K^{*1} * t_1 = 2 * 1013 = 2026$

Se puede comprobar que el tiempo es más o menos correcto.

--> SumaVector3:

$t_1 = 1199$  para  $n_1 = 48$

$t_2 = X$  para  $n_2 = 96$

$K = 96/48 = 2$

$t_2 = K^{*1} * t_1 = 2 * 1199 = 2398$

Se puede comprobar que el tiempo es más o menos correcto.

Usando uso de las fórmulas de divide y vencerás:

--> SumaVector2 (caso sustracción):

$a = 1$

$b = 1$

$k = 0$

Por tanto tenemos que  $O(n)$ .

--> SumaVector3 (caso división):

$a = 2$

$b = 2$

$k = 0$

Por tanto tenemos que  $O(n)$ .

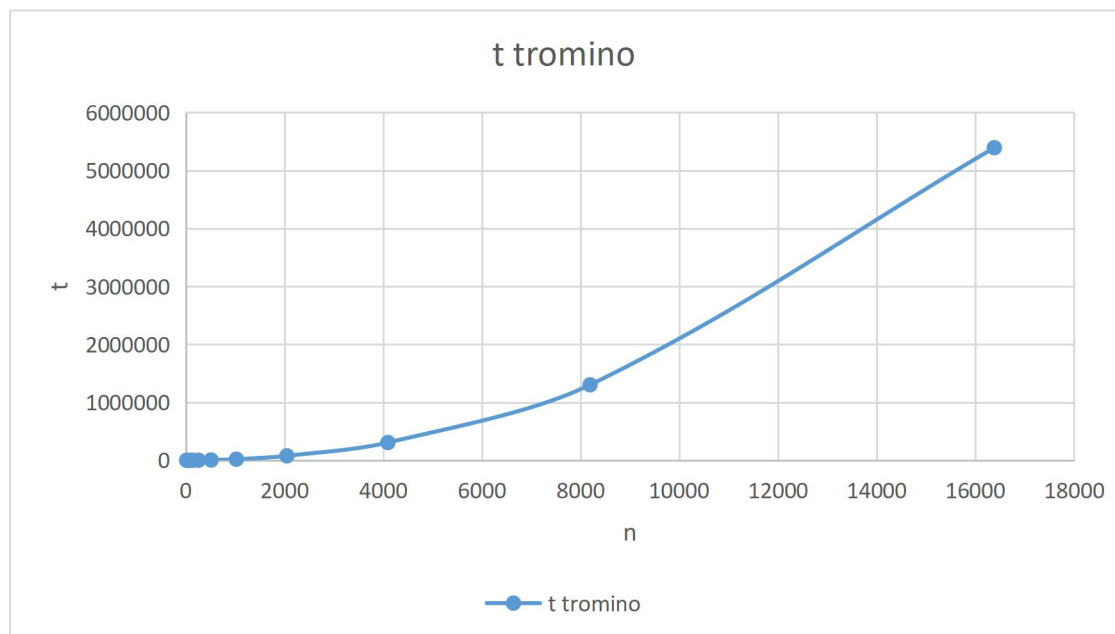
## TIEMPOS TROMINÓ

En el caso del trominó los tiempos medidos de la ejecución para diferentes tamaños de tablero son:

n	t tromino
16	11
32	16
64	66
128	279
256	1078
512	4474
1024	18588
2048	77132
4096	306632
8192	1302413
16384	5394073

n veces = 10000

La gráfica correspondiente es la siguiente:





Ahora usando el análisis que llevamos haciendo durante toda la práctica es útil.

Suponiendo que tenemos una complejidad de  $O(n^2)$  en la resolución del problema del trominó. Haciendo uso de los tiempos podemos comprobar si dicha complejidad es correcta.

Por tanto usando los tiempos tenemos que:

$$t_1 = 4474 \text{ para } n_1 = 512$$

$$t_2 = X \text{ para } n_2 = 1024$$

$$K = 1024/512 = 2$$

$$t_2 = K^2 * t_1 = 2^2 * 4474 = 17896$$

Que se asemeja mucho al valor medido en la ejecución (18588). También hacemos uso de las fórmulas de divide y vencerás, ya que es un algoritmo claramente que sigue la filosofía de esta práctica.

Por lo tanto teniendo:

$$a = 4$$

$$b = 2$$

$$k = 0$$

Tenemos un resultado final de  $O(n^2)$ .

En conclusión la resolución al problema del trominó implementada sigue una complejidad cuadrática.