Tarea Practica 1 Iker Núñez UO306425

2)Toma de tiempos de ejecución

Como currrentTimeMillis() devuelve un entero de 64 bits (long) que es el milisegundo en el que vive el ordenador actualmente. Luego, si quieres medir un tiempo de ejecución, no tienes que hacer más que llamar al método antes y después de ejecutarlo y restar después – antes.

La cuenta a cero se puso hace más de 50 años.

¿Cuántos años más podremos seguir utilizando este procedimiento para medir tiempos?

Bueno: como devuelve un entero de 64 bits, podremos representar 2^64 números, que son 18,446,744,073,709,551,616 números. Ahora, podemos ejecutar Vector2.java para algún n grande. Yo en mi caso escogí n = 100000000 y me salió t1 = 1770276323069.

Ahora podemos restar t1 – 2^64 y nos saldrá el número de años, pero le tendremos que restar 56 años, porque la cuenta a cero se puso hace 56 años.

Para no poner resultados diferentes a los obtenidos en clase, pongo el numero de años que nos salieron en clase, pero no me acuerdo del n que utilizamos:

5.8 * 10^8 – 56 años.

¿Por qué a veces los tiempos de ejecución salen cero?

Esto puede ser por varias razones. Si estamos ejecutando Java con optimización, puede ser porque el compilador JIT se salta algunas instrucciones y recupera algunos resultados de ejecuciones anteriores, luego tarda menos.

En nuestro caso, estamos ejecutando Java sin optimización. La respuesta es que a veces, al ejecutar un programa en Java, el recolector de basura se ejecuta. Esto hace que se pausen todos los procesos, y por esta razón, los tiempos de ejecución salen cero.

¿A partir de que n se empiezan a obtener tiempos fiables?

Los tiempos de ejecución a veces pueden ser menores que 50 ms dependiendo del n que le pases. Estos tiempos serán descartados por falta de fiabilidad.

En el caso del programa Vector2.java, se empiezan a obtener tiempos fiables a partir de: 5.500.000 en mi caso desde el ordenador de casa, para el que se obtiene un tiempo de 51ms

```
PS C:\Users\IkerNuevo\Desktop\alg_Nu-ezMeanaIkerUO306425\p11> java -Xint p11.Vector2 100000000
t1=1770276323069  *** t2=1770276324017
n= 100000000    Tiempo metodo suma = 948
Resultado de la suma de elementos = 287119
PS C:\Users\IkerNuevo\Desktop\alg_Nu-ezMeanaIkerUO306425\p11> java -Xint p11.Vector2 1500000
t1=1770276850789  *** t2=1770276850803
n= 1500000      Tiempo metodo suma = 14
Resultado de la suma de elementos = 41180
PS C:\Users\IkerNuevo\Desktop\alg_Nu-ezMeanaIkerUO306425\p11> java -Xint p11.Vector2 1990000
t1=1770276869496  *** t2=1770276869513
n= 1990000      Tiempo metodo suma = 17
Resultado de la suma de elementos = -35769
PS C:\Users\IkerNuevo\Desktop\alg_Nu-ezMeanaIkerUO306425\p11> java -Xint p11.Vector2 10000000
t1=1770276879483  *** t2=1770276879578
n= 10000000     Tiempo metodo suma = 95
Resultado de la suma de elementos = -40015
PS C:\Users\IkerNuevo\Desktop\alg_Nu-ezMeanaIkerUO306425\p11> java -Xint p11.Vector2 7000000
t1=1770276903111  *** t2=1770276903176
n= 7000000      Tiempo metodo suma = 65
Resultado de la suma de elementos = -140129
PS C:\Users\IkerNuevo\Desktop\alg_Nu-ezMeanaIkerUO306425\p11> java -Xint p11.Vector2 5000000
t1=1770276912580  *** t2=1770276912627
n= 5000000      Tiempo metodo suma = 47
Resultado de la suma de elementos = 104830
PS C:\Users\IkerNuevo\Desktop\alg_Nu-ezMeanaIkerUO306425\p11> java -Xint p11.Vector2 5500000
t1=1770276920012  *** t2=1770276920063
n= 5500000      Tiempo metodo suma = 51
Resultado de la suma de elementos = 497
PS C:\Users\IkerNuevo\Desktop\alg_Nu-ezMeanaIkerUO306425\p11>
```

Adjunto una tabla con los tiempos para distintos n hecha en clase:

Vector 2					
n	100000	10000000	50000000	100000000	500000000
t(ms)	5	47	239	481	2412

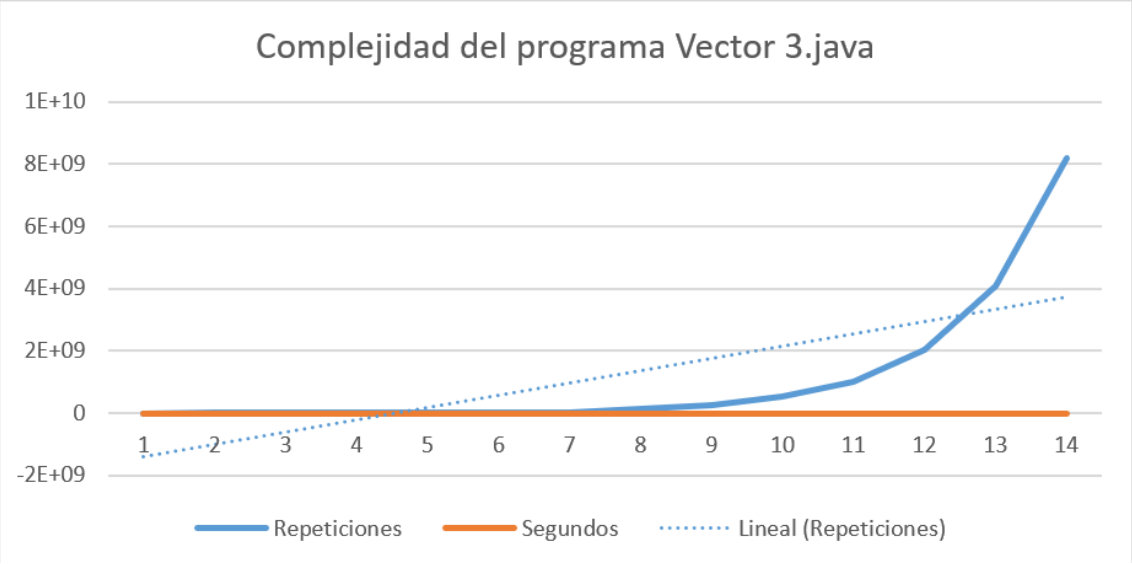
3) Crecimiento del tamaño del problema

A continuación, se adjunta una tabla con los tiempos de ejecución para distintos n SIN OPTIMIZACION como las ejecuciones anteriores:

Vector 3										
n	100000		10000000		50000000		100000000		500000000	
t(ms)	n	tTiempo	n	tTiempo	n	tTiempo	n	tTiempo	n	tTiempo
	10000	0	10000	0	10000	1	10000	0	10000	0
	20000	0	20000	0	20000	0	20000	0	20000	0
	40000	1	40000	0	40000	0	40000	0	40000	0
	80000	0	80000	0	80000	0	80000	1	80000	0
	160000	0	160000	1	160000	2	160000	2	160000	2
	320000	1	320000	2	320000	3	320000	2	320000	2
	640000	3	640000	4	640000	5	640000	6	640000	5
	1280000	6	1280000	6	1280000	18	1280000	12	1280000	11
	2560000	12	2560000	13	2560000	24	2560000	26	2560000	25
	5120000	24	5120000	27	5120000	47	5120000	63	5120000	56
	10240000	49	10240000	50	10240000	101	10240000	99	10240000	93
	20480000	98	20480000	99	20480000	210	20480000	182	20480000	190
	40960000	197	40960000	209	40960000	387	40960000	389	40960000	402
	81920000	393	81920000	408	81920000	771	81920000	748	81920000	763

Si cogemos los tiempos de una columna, los sacamos a una tabla y representamos la grafica en una tabla, podemos ver la complejidad del algoritmo de forma gráfica. Realizandolo para n = 100000 por ejemplo:

Segundos	Repeticiones
0	10000
0	20000
1	40000
0	80000
0	160000
1	320000
3	640000
6	128000000
12	256000000
24	512000000
49	1024000000
98	2048000000
197	4096000000
393	8192000000

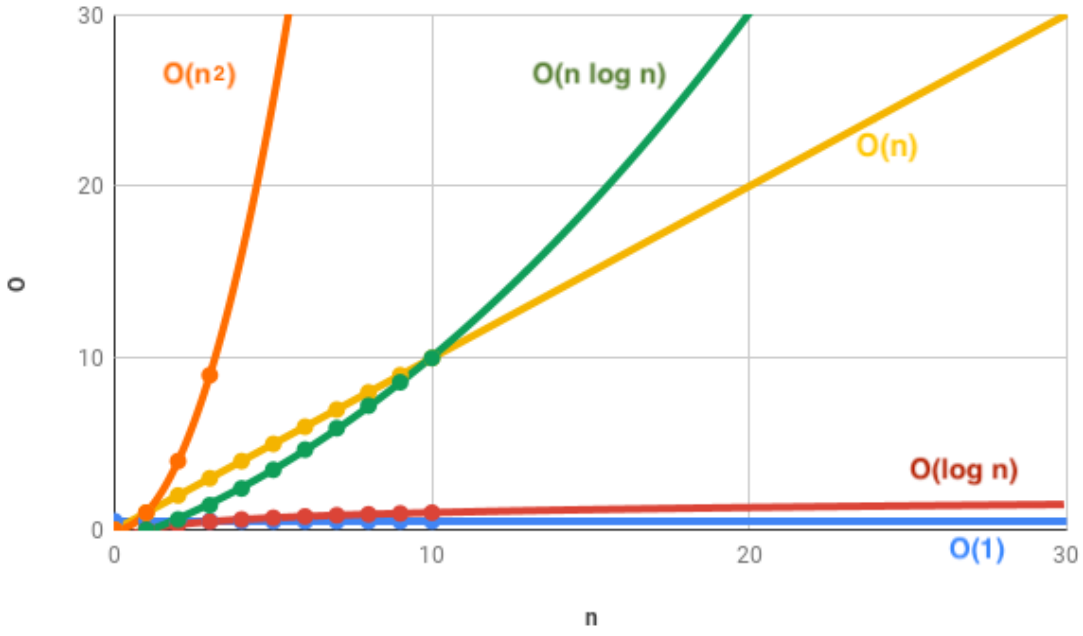


Como vemos, los tiempos que salen son solo los que se tardo en realizar el algoritmo suma, que tiene complejidad 0(n):

```
public static void main (String arg [] )
{
    long t1,t2;

    System.out.println (x: "n   tTiempo");
    for ( int n=10000; n<= 81920000; n*=2) // n se va incrementando (*2)
    {
        System.out.print (n+"\t");
        v = new int [n] ;
        Vector1.rellena (v);

        // Medida del tiempo de la operacion suma()
        t1=System.currentTimeMillis();
        int s=Vector1.suma (v);
        t2=System.currentTimeMillis();
        System.out.println ((t2-t1)+"\t");

        } // fin de for de n del problema

    System.out.println(x: "\nFin de la medicion de tiempos *****");
```

Luego, si nos fijamos en la línea de tendencia que sale en la gráfica anterior, vemos que la tendencia es lineal. Luego, los resultados pueden ser coherentes con la complejidad del algoritmo:

```
/* Este metodo suma los elementos de un vector y devuelve el resultado
*/
public static int suma (int[]a)
{
    int s=0;
    int n= a.length;
    for (int i=0;i<n;i++) s=s+a[i];
    return s;

}  // fin de suma
```



La inclinacion en nuestra grafica es menos notable, pero es por los valores escogidos y la escala.

4) Toma de tiempos pequeños

Ejecuciones de Vector4.java:

1		10		100		1000		100000		10000000	
n	Tiempo	n	Tiempo	n	Tiempo	n	Tiempo	repeticiones = 100000		n	Tiempo
10000	0	10000	1	10000	14	10000	103	n	Tiempo	10000	89715
20000	0	20000	1	20000	17	20000	191	10000	6619	20000	202705
40000	1	40000	6	40000	38	40000	461	20000	13171	40000	661730
80000	1	80000	9	80000	73	80000	848	40000	26334	80000	1254631
160000	2	160000	18	160000	147	160000	1687	80000	53584	160000	2242497
320000	3	320000	28	320000	323	320000	3281	160000	106730	320000	4818522
640000	6	640000	72	640000	670	640000	6469	320000	210502	640000	10210934
1280000	12	1280000	127	1280000	1288	1280000	12958	640000	425768	1280000	25527904
2560000	27	2560000	283	2560000	4479	2560000	26837	1280000	835628	2560000	117180612
5120000	52	5120000	508	5120000	6985	5120000	52089	2560000	8733776		
10240000	159	10240000	1029	10240000	18683	10240000	101812	5120000	5401252		
20480000	271	20480000	1992	20480000	24603	20480000	269438	10240000	6841835		
40960000	462	40960000	3980	40960000	42466	40960000	348076	20480000	13674182		
81920000	1474	81920000	8163	81920000	142541	81920000	548334				

En mi caso, para n = 100000 y n = 1000000 corte la ejecución para los últimos números ya que tardaban mas de 7h en hacer 2 iteraciones. Sin embargo, si miramos la tabla, podemos ver que para n = 100000 frente a n = 1000 se multiplican x50 e incluso por mas de 60 para algún caso los tiempos:

13674182 / 269438 = 50,75

6841835 / 101812 = 67,20

Para n = 1000000 frente a n = 100000 pasa algo similar:

25527904 / 835628 = 30,55

Pero en menor proporción en este caso.

Las ejecuciones de este programa fueron cortadas al llegar a n altos debido al gran tiempo que tardaron en resolverse, el cual se puede ver en las capturas.

¿Qué pasaría si multiplicamos el tamaño del problema por 2?

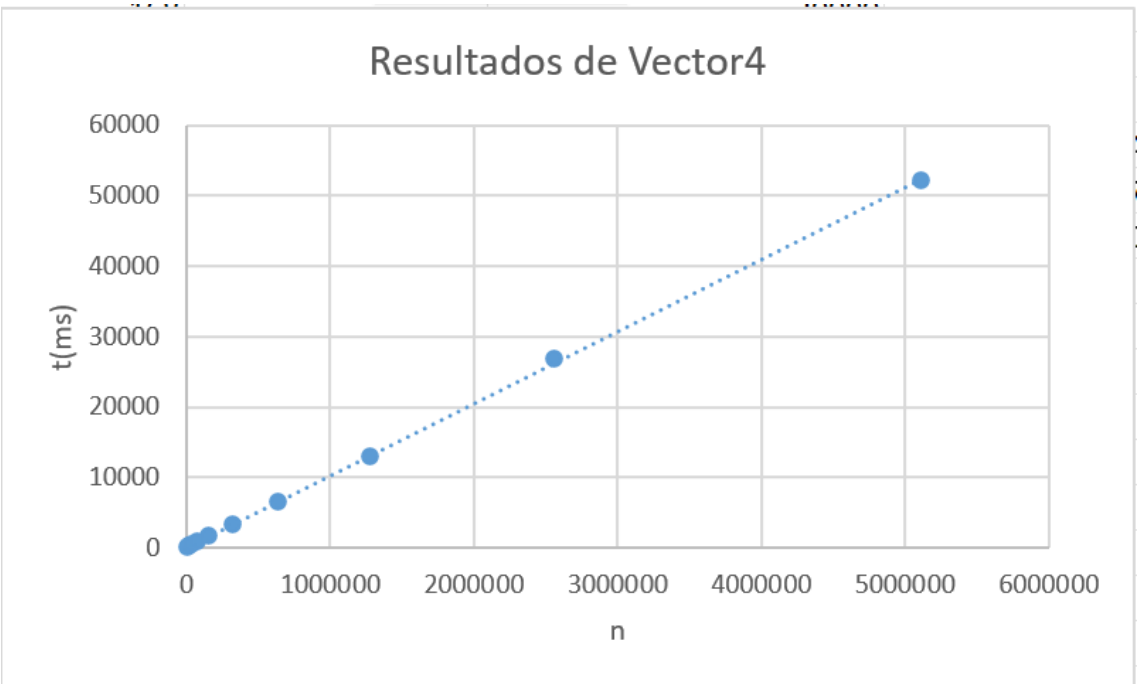
De acuerdo con la formula vista en la teoría, sabemos que si para un algoritmo de complejidad 0(lo que sea) y para n1 = x tarda un tiempo t1 = y segundos, entonces, si aumentamos el tamaño del problema en k, entonces t2 = k * t1.

¿Qué pasa si en vez de por 2 multiplicamos por otra constante (por ejemplo, probar con k = 3 y k = 4?

Pasaría lo mismo que el caso anterior donde k = 2, esta vez para k = 3 y k = 4.

¿Son los tiempos obtenidos razonables con la complejidad lineal del algoritmo?

Para comprobar esto, podemos representar los tiempos obtenidos en una gráfica y ver como sale la recta. En este caso, cojo, por ejemplo, los tiempos obtenidos para n = 1000:



Vemos que para todos los puntos, sale la tendencia esperada por su complejidad (0(n)).

En la siguiente parte, se pide elaborar 3 programas mas para medir los tiempos de otros algoritmos. Tras obtener los tiempos, se piden elaborar algunas tablas.

En todos los casos, pongo n = 1000 por ejemplo:

n	TiempoSuma	TiempoMaximo
10000	103	96
20000	191	168
40000	461	341
80000	848	682
160000	1687	1384
320000	3281	2712
640000	6469	5357
1280000	12958	10843
2560000	26837	21626
5120000	52009	43836
10240000	101812	88097
20480000	269438	18141411
40960000	348076	504529
81920000	548334	1600746

Para estas dos ejecuciones, use n = 10 ya que Vector6 tardaba mucho en ejecutarse:

n	TiempoCoincidencias1	TiempoCoincidencias2
10000	8359	2
20000	41484	4
40000	136952	5
80000	757538	12
160000	4411314	20
320000	17320862	39
640000	120958434	77
1280000	Fdt	164
2560000	fdt	339
5120000	fdt	660
10240000	fdt	1321
20480000	fdt	2708
40960000	fdt	5273
81920000	fdt	10829

¿Los resultados son coherentes con las complejidades de los algoritmos?

Primero, hemos de mirar cuales son las complejidades:

Vector 4 -> algoritmo de suma -> 0(n)

Vector 5 -> algoritmo de máximo -> 0(n)

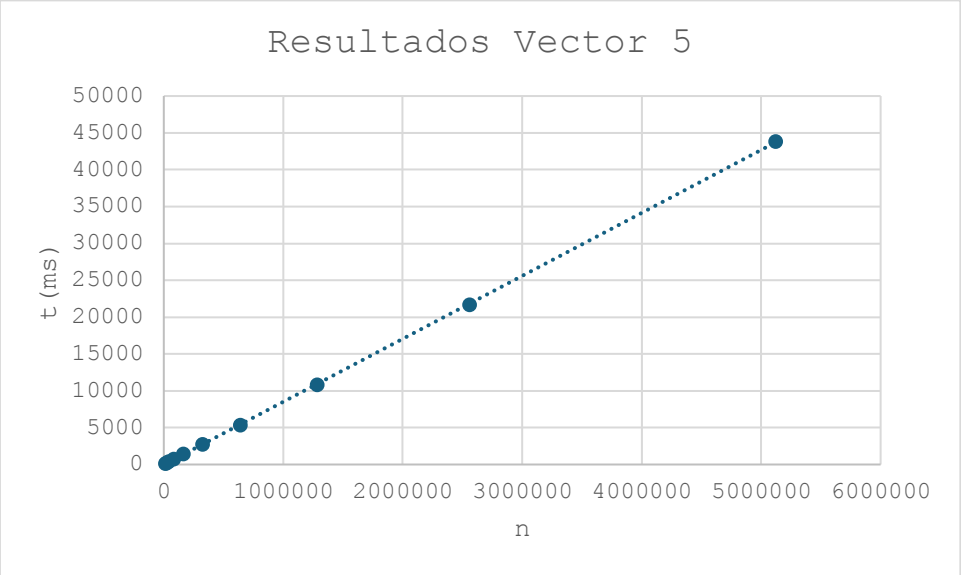
Vector 6 -> algoritmo coincidencias 1 -> 0(n^2)

Vector 7 -> algoritmo coincidencias 2 -> 0(n)

Los resultados del vector 4 ya vimos antes que si eran coherentes, como cogí los mismos resultados no volveré sobre estos.

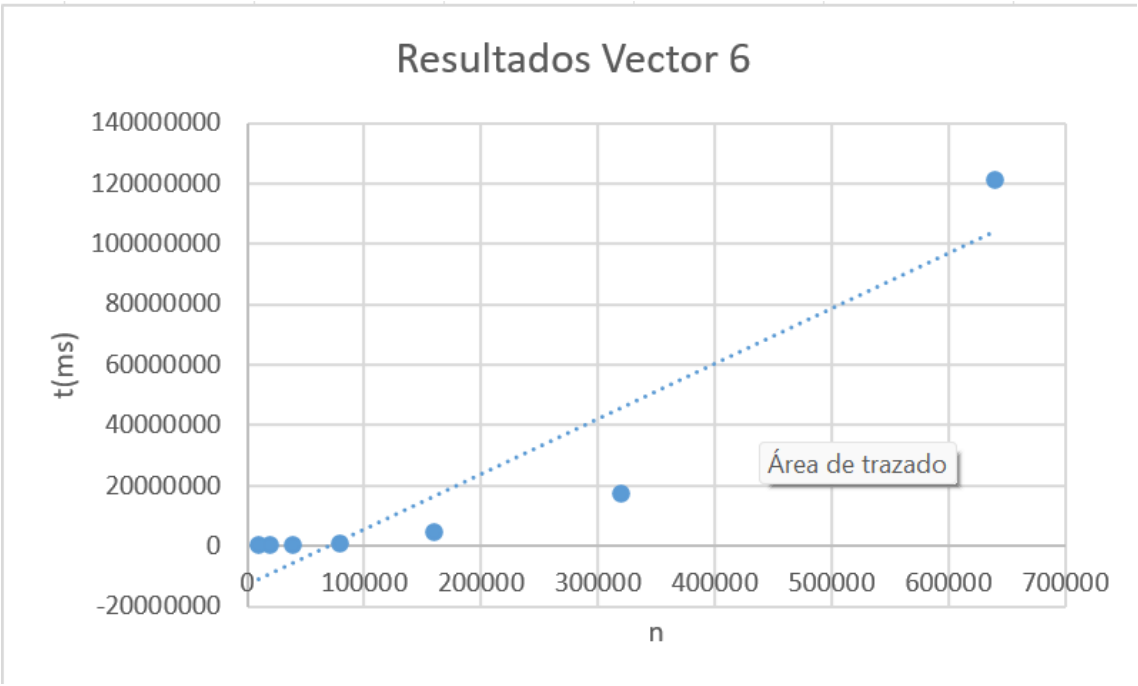
Para los resultados de Vector5:

En la siguiente gráfica, podemos ver los resultados visualmente:



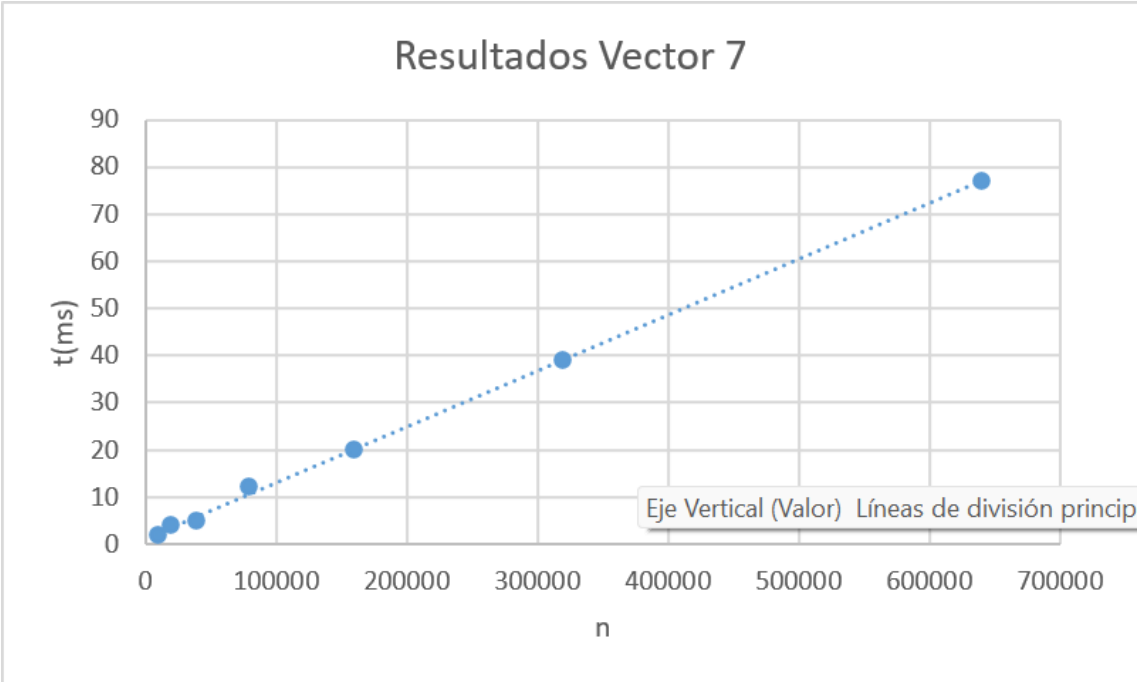
Si nos fijamos en la línea de tendencia, vemos que la tendencia es lineal, luego los resultados serían coherentes.

Para los resultados del Vector 6:



En este caso, vemos que la línea de tendencia tiene mayor inclinación que en la gráfica anterior. Sin embargo, al ser una complejidad cuadrática, quizás debería ser mas inclinada aún. Por lo tanto, podemos decir que los resultados no son tan coherentes.

Para los resultados de Vector7:



Esta grafica certifica que los resultados de Vector6 no fueron los esperados, ya que la línea de tendencia tiene la misma inclinación y la complejidad de este algoritmo es 0(n), por tanto, para Vector7, los resultados si pueden ser coherentes, pero para Vector6 no.

Las características del ordenador donde he medido los tiempos son:

Apartados 1 y 2: Ordenador de clase

Procesador: 12th Gen Intel® Core™ i5-12400 (2.50GHz)

Memoria RAM: 16,0 GB (15,8 GB usable)

Resto de las mediciones: Ordenador personal

Procesador: 13th Gen Intel(R) Core(TM) i7-13700H (2.40 GHz)

Memoria: 16,0 GB (15,7 GB usable)