PRACTICA 3 ALGORITMIA

Nahiara Sánchez García

ÍNDICE

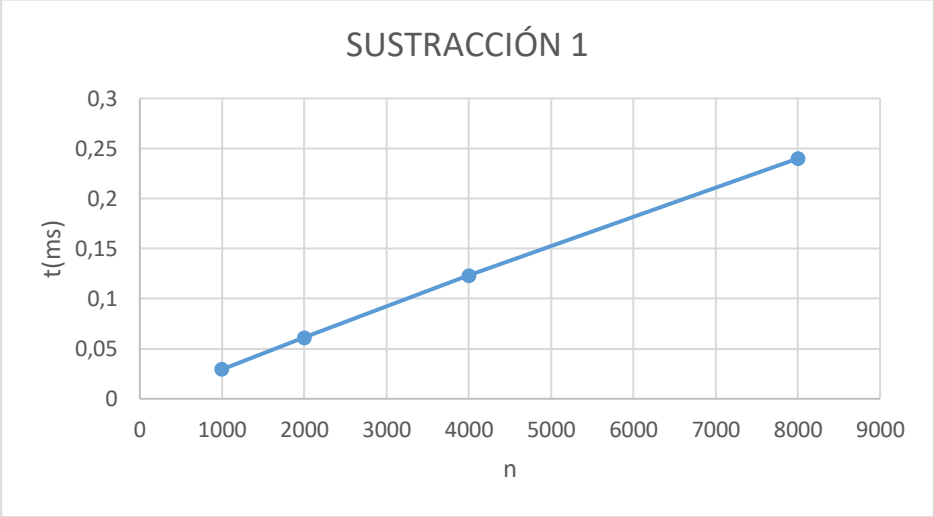
Contenido

Objetivo:

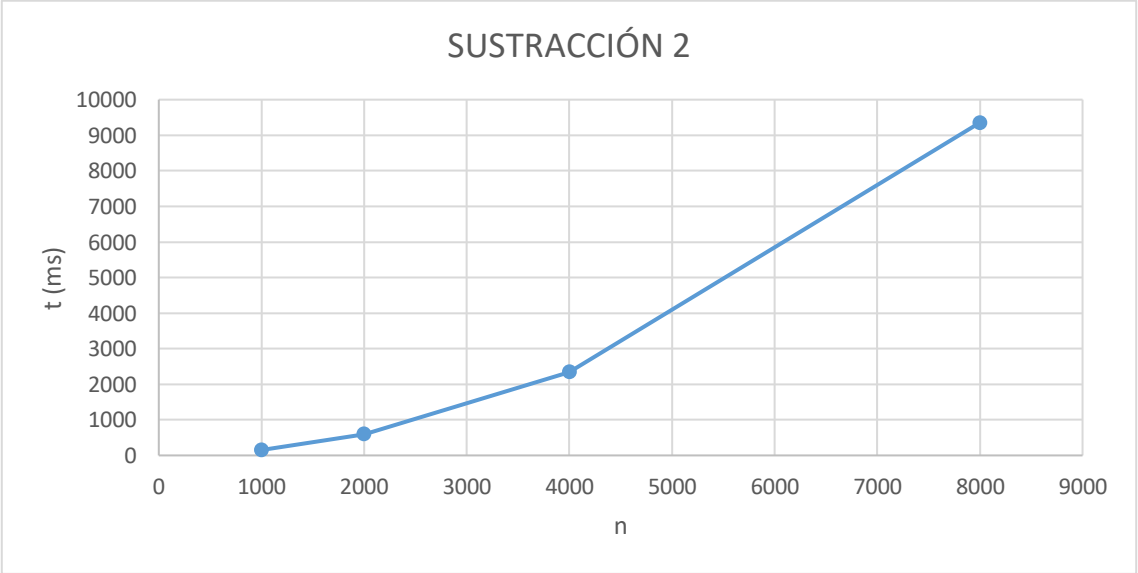
El objetivo de esta práctica es estudiar la complejidad temporal y el tiempo de ejecución de distintos algoritmos recursivos Divide y Vencerás (con sustracción y con división). También se dará una solución a un problema concreto con este tipo de algoritmos, se medirán tiempos de ejecución de dicho algoritmo y se analizarán los resultados obtenidos. Hay que tener en cuenta que si es por sustracción, si a = 1 la complejidad será $O(n^{k+1})$, en caso contrario, la complejidad será $O(a^{n/b})$. Por otra parte, si se realiza por división: si $a < b^k$ la complejidad es $O(n^k)$; si $a = b^k$ la complejidad es $O(n^k \log(n))$; si $a > b^k$ entonces la complejidad será $O(n^{\log_b a})$.

EJERCICIO 1

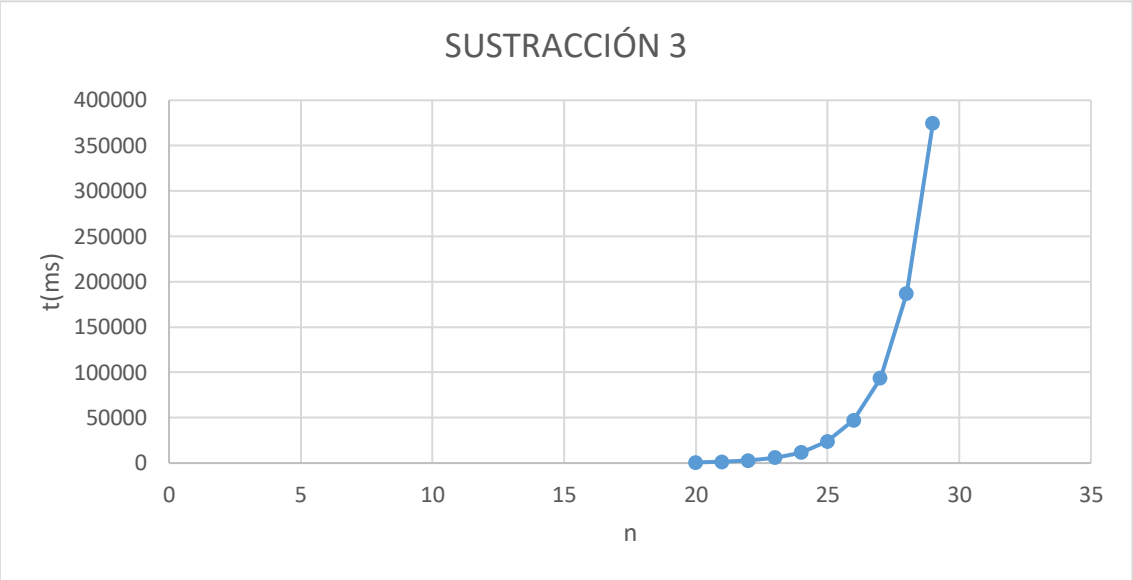
1. Tras analizar las complejidades de las clases Sustracción1, Sustracción2 y Sustracción3 los resultados obtenidos son los siguientes:
   - Sustracción 1: Teniendo en cuenta que a = 1, b = 1 y k = 0 la complejidad será $O(n^{k+1})$, es decir, $O(n^1)$, que es **O(n).**
     Teniendo en cuenta que tiene una complejidad lineal, se verá a continuación si la gráfica cumple con lo esperado:



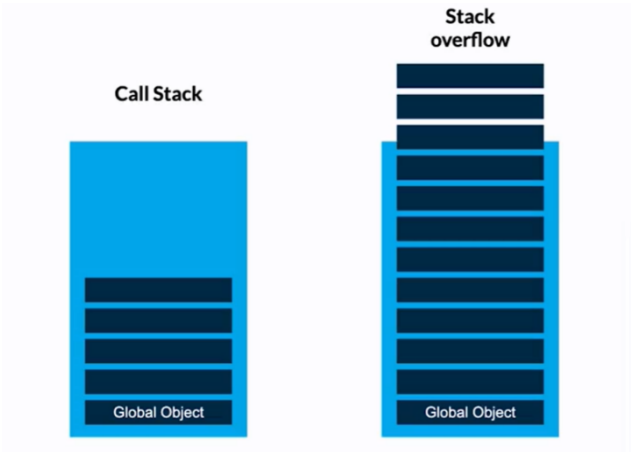
   - Sustracción 2: Teniendo en cuenta que a = 1, b = 1 y k = 1 la complejidad será $O(n^{k+1})$, es decir, **$O(n^2)$.**Si se observa la siguiente gráfica se puede ver que los tiempos obtenidos son los esperados de una complejidad cuadrática:



- Sustracción 3: Teniendo en cuenta que a = 2, b = 1 y k = 0 la complejidad será $O(a^{n/b})$, es decir, **$O(2^n)$.**
  Si se observa la gráfica se puede ver que cumple lo esperado de una función exponencial.



Los métodos de las clases Sustraccion1 y Sustraccion2 dejan de dar tiempos a partir de n = 8000. Esto sucede ocasionando un error: `java.lang.StackOverflowError`. Es así porque en cada llamada a un método se utiliza una parte de la memoria, denominada stack. Si se hacen muchas llamadas puede ocurrir que ese espacio se llene y genera un stack overflow. Básicamente ocurre un desbordamiento de tareas. Algunos de los motivos pueden ser: bucles infinitos, funciones recursivas sin control, cambios de estado continuo…



A continuación, se quiere determinar cuantos años tardaría en finalizar la ejecución de Sustracción 3, pues ya se vio anteriormente que la complejidad de este método crece exponencialmente.

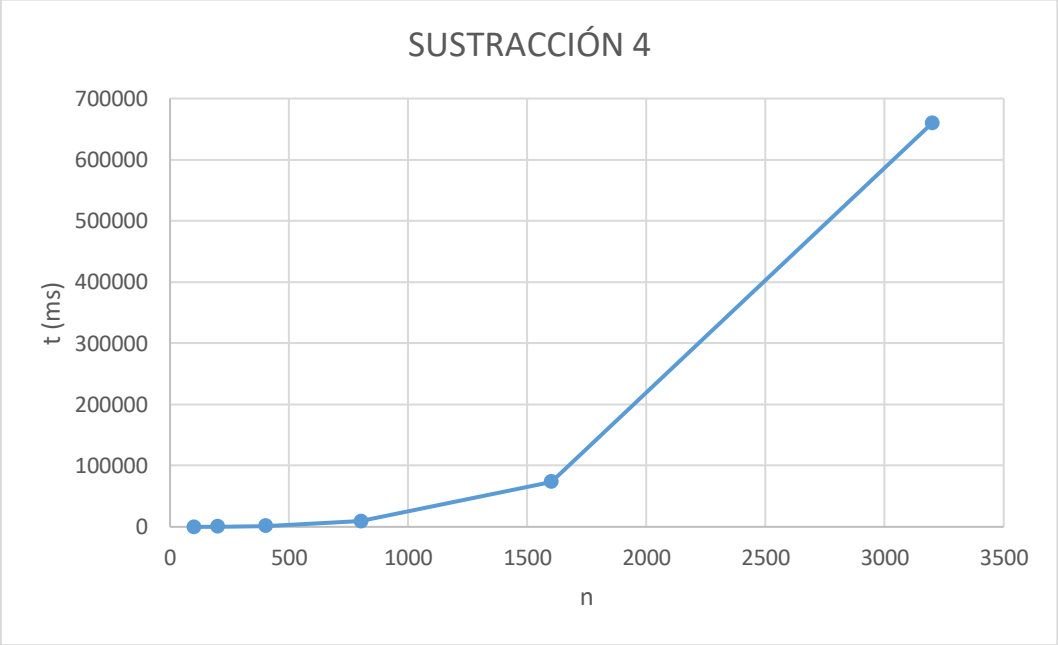
- El tiempo expresado en milisegundos es: 8,50856E+20 ms.
- El tiempo expresado en años es: 26.980.469.001 años.

Es decir, el método Sustracción 3 tarda millones de años en ejecutarse para n = 80.

Para continuar con la práctica se implementan las clases Sustracción 4 y Sustracción 5 con complejidades $O(n^3)$ y $O(3^{n/2})$ respectivamente.

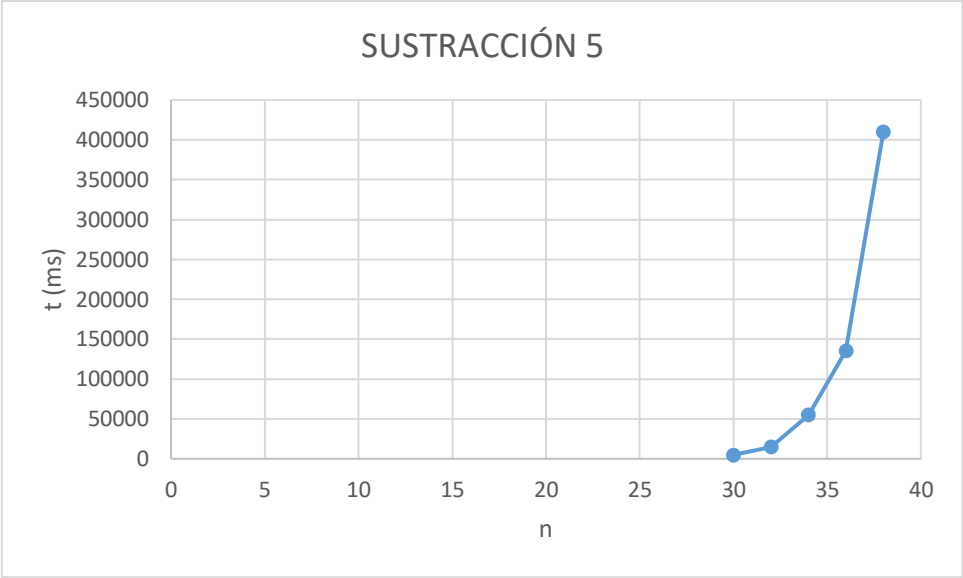
- Sustracción 4: Este algoritmo, como bien se especificó anteriormente, tiene una complejidad $O(n^3)$. Como se ve, los datos cumplen lo esperado de una complejidad de este tipo.

n	Algoritmo Sustracción 4
100	1,88
200	175
400	1195
800	9095
1600	73426
3200	659954



- Sustracción 5: Este algoritmo tiene una complejidad $O(3^{n/2})$. Como se observa, los datos obtenidos tanto en la tabla como en la gráfica son los esperados.

n	Algoritmo Sustracción 5
30	4930
32	14857
34	54850
36	135243
38	409864

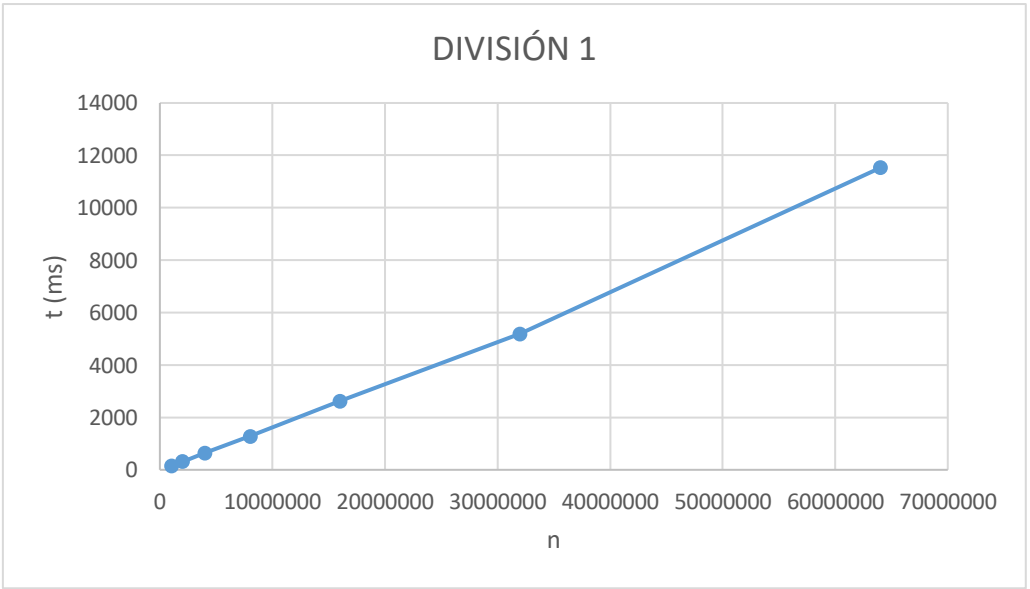


¿Cuánto tardará en finalizar la ejecución Sustracción 5 para n = 80?

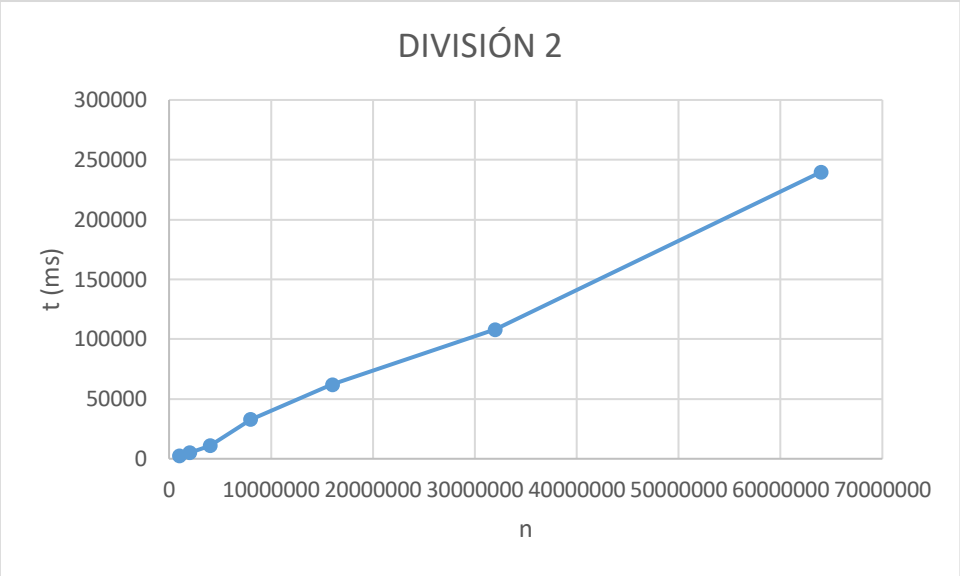
- Tardará 4,17713E+15 ms
- Expresado en años será: 132.456,0136 años

EJERCICIO 2

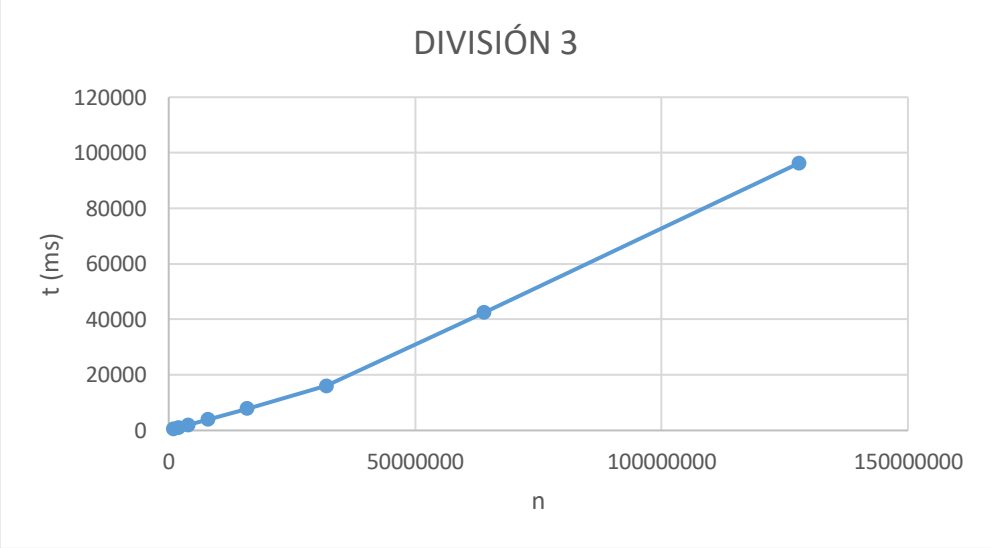
2. Ahora se realizarán algoritmos Divide y Vencerás por división, cuyas fórmulas sobre el cálculo de complejidad se explicaron al principio del documento.
   - <u>División 1:</u> Como $a < b^k$, la complejidad de este algoritmo es O(n). Como se logra ver en la gráfica, los datos obtenidos coinciden a la perfección con lo que se estimaba.



- <u>División 2:</u> Como a = b^k, la complejidad es O(n log (n)). Se puede comprobar que la gráfica colocada a continuación cumple con lo esperado de dicha complejidad.

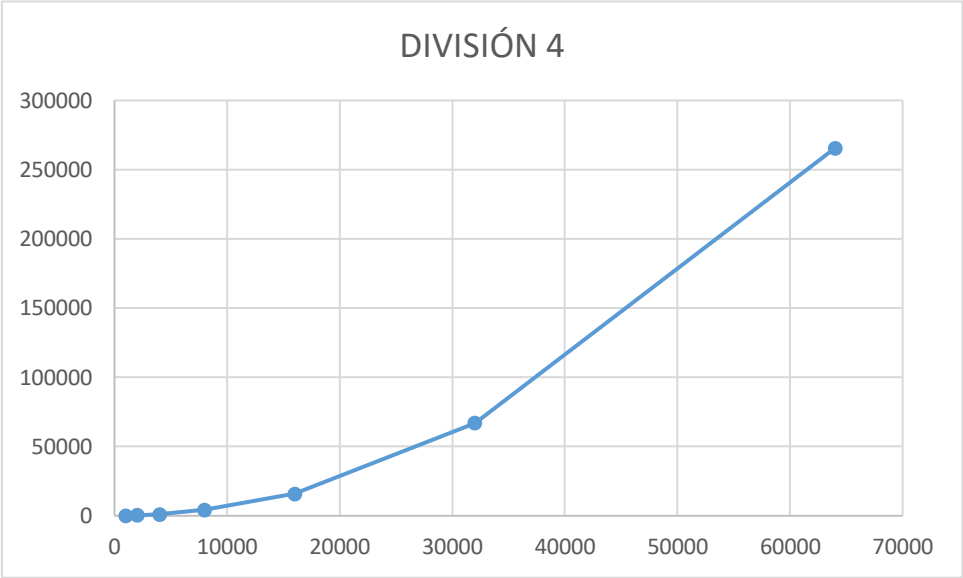


- <u>División 3:</u> Como a > b^k, la complejidad es O(n $\log_2(2)$). Este algoritmo obtendrá tiempos bastante menores que en los casos anteriores. La gráfica es:



Tras implementar la clase Division4 con una complejidad **O(n^2)** mediante **a < b^k**, los tiempos que se obtienen son los siguientes:

n	t (ms)
1000	70
2000	308
4000	1109
8000	4257
16000	15944
32000	66878
64000	265544

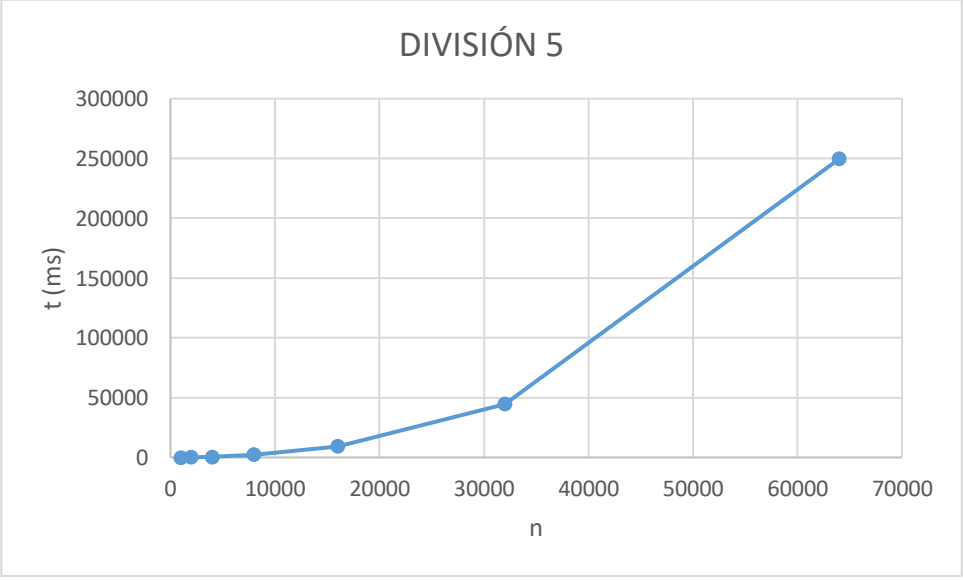


Como se puede ver, los datos de la tabla y la gráfica concuerdan con una complejidad $O(n^2)$.

A continuación, se quiere implementar División5 con una complejidad **$O(n^2)$,** pero esta vez de tipo **$a > b^k$**. Los tiempos que se obtienen son los siguientes:

n	t (ms)
1000	39,9
2000	167
4000	589
8000	2254
16000	9220
32000	44529
64000	249682

Y su correspondiente gráfica:



Se puede ver que los datos obtenidos corresponden a una complejidad $O(n^2)$.

EJERCICIO 3

3. Este ejercicio trata de analizar la complejidad de los diversos algoritmos en las clases VectorSuma y Fibonacci, para después ejecutarlas y realizar las mediciones de tiempo.

- **VectorSuma:** Esta clase trata de sumar los elementos de un vector de tres maneras diferentes.
    1) Sum1: Tiene una complejidad O(n) y está implementado de manera iterativa.
    2) Sum2: Tiene una complejidad O(n) y está implementado con divide y vencerás por sustracción.
    3) Sum3: Tiene una complejidad O(n) y está implementado con divide y vencerás por división.

Método 1 – Sum1	Método 2 – Sum2	Método 3 – Sum3
720	1715	3486

Como se puede ver, aunque todos los métodos tienen una complejidad O(n), hay mucha diferencia de tiempo entre ellos. Esto es normal, pues la complejidad solo indica que el tiempo de ejecución del algoritmo crece de formal lineal, pero no indica nada sobre la tasa de crecimiento exacta.
Algunas razones que afectan a esto podrían ser: la implementación y eficiencia del algoritmo, diferencias en el factor de crecimiento…
Por tanto, O(n) solo indica cómo crece el tiempo de ejecución de un algoritmo a medida que crece el tamaño de la entrada.

- **Fibonacci:** Esta clase trata de resolver Fibonacci de cuatro maneras diferentes. Los tiempos dados son en nanosegundos.
    1) Fib1: Tiene una complejidad O(n) implementado de manera iterativa.
    2) Fib2: Tiene una complejidad O(n) implementado de manera iterativa.
    3) Fib3: Tiene una complejidad O(n) implementado mediante divide y vencerás por sustracción.
    4) Fib4: Tiene una complejidad $O(1{,}6^n)$ implementado mediante divide y vencerás por sustracción, es decir, es exponencial.

Método 1 – Fib1	Método 2 – Fib2	Método 3 – Fib3	Método 4 – Fib4
294	468	656	4942

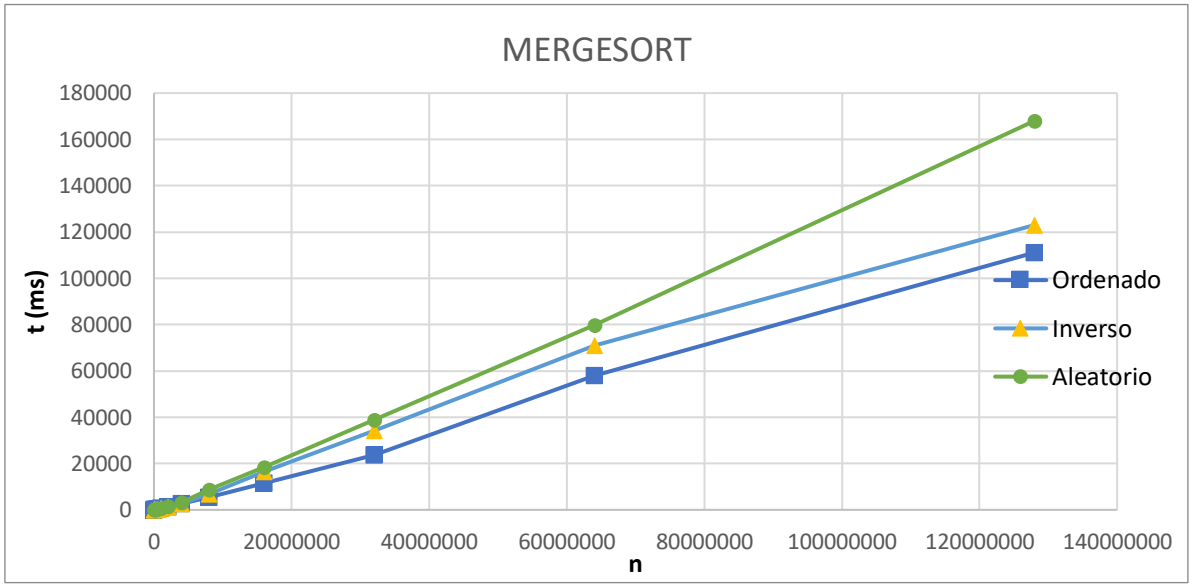
Como se puede ver el método 4 obtiene un tiempo bastante superior al resto, debido a su complejidad exponencial. Sin embargo, el resto tienen la misma complejidad y sus tiempos son diferentes.
Esto es así debido a lo que se explicó anteriormente. La complejidad sólo indica cómo crece el tiempo de ejecución, pero esto no quiere decir que los tiempos vayan a ser los mismos.

EJERCICIO 4

4. Tras implementar la clase Mezcla, encargada de ordenar los elementos de un vector, y MezclaTiempos para las mediciones de tiempos, se observan los siguientes datos:

n	t ordenado	t inverso	t aleatorio
31250	FdF	FdF	FdF
62500	FdF	FdF	FdF
125000	63	61	80
250000	129	133	184
500000	300	287	317
1000000	603	610	678
2000000	1271	1280	1458
4000000	2685	2697	3120
8000000	5443	6687	8838
16000000	11436	16381	18525
32000000	23781	34297	38959
64000000	57863	70847	79726
128000000	111125	123024	168035

Y la correspondiente gráfica:



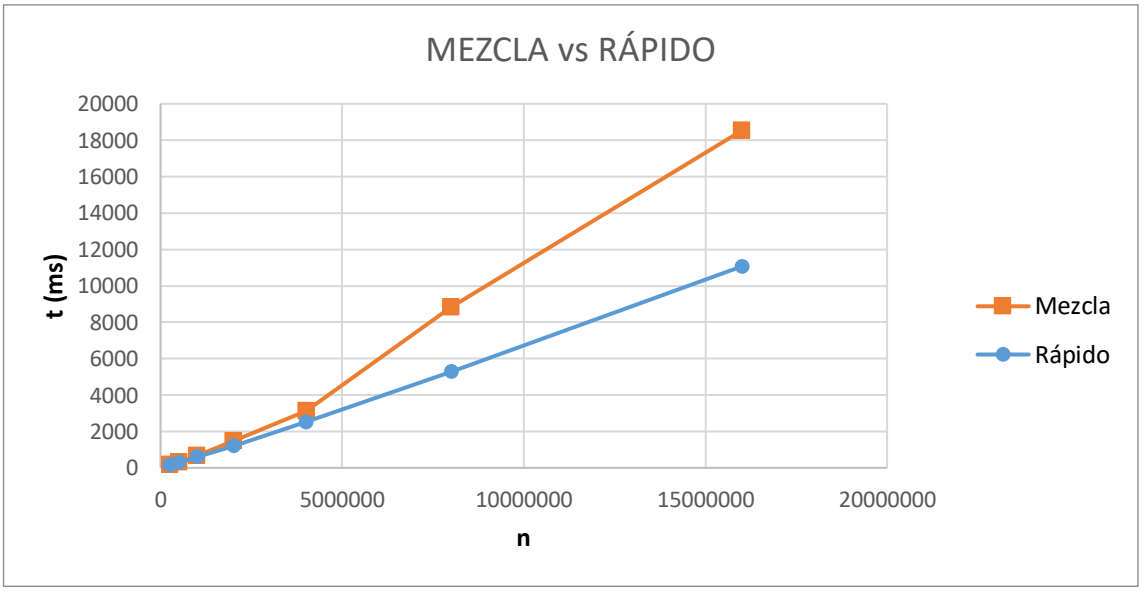
Teniendo en cuenta que este algoritmo presenta una complejidad O(log(n)) para la división del array y O(n) para la fusión de los elementos, la complejidad total del algoritmo es O(n log(n)). Por lo que se puede decir que los datos concuerdan con lo esperado.

Una vez realizadas las mediciones hasta que están fuera de tiempo (superior al minuto) se observa lo siguiente: al llegar a un tamaño **n =128000000** salta el error `java.lang.OutOfMemoryError: Java heap space`. Este error indica que el programa se ha quedado sin memoria en el heap (región de memoria en la que se almacenan los objetos y sus referencias) de Java, es decir, el programa ha usado demasiada memoria.

A continuación, se pretende comparar este algoritmo con el ya visto en sesiones anteriores, rápido. Hay que tener en cuenta que este estudio de rápido se realiza escogiendo como pivote el elemento central, por lo que en ningún caso se dará el caso peor $O(n^2)$, si no que la complejidad será O(n log(n)). Por esto mismo, puede ser que el algoritmo rápido sea ligeramente más rápido (valga la redundancia) que el algoritmo mezcla.

n	t1 (mezcla en ms)	t2 (rápido en ms)	t1 / t2
250000	184	136	1,35294118
500000	317	281	1,12811388
1000000	678	592	1,14527027
2000000	1458	1204	1,21096346
4000000	3120	2510	1,24302789
8000000	8838	5271	1,67672168
16000000	18525	11067	1,67389536

Como se puede observar, la **constante** que sale como comparación de los dos algoritmos es **1.** Esto quiere decir que es tanto mejor el algoritmo colocado en el denominado que el colocado en el numerador.
Se muestra a continuación la gráfica:



Al observar la gráfica se logra ver que los tiempos tomados por el algoritmo rápido son menores que los tiempos tomados con el algoritmo mezcla.
Nuevamente es necesario recordar que esto es así siempre y cuando en el algoritmo rápido se escoja como pivote el elemento central, pues en otros casos esta afirmación podría ser falsa.