Ejercicio 1:

Tras analizar la complejidad de las tres clases anteriores (Sustraccion1.java, Sustraccion2.java y Sustraccion3.java), no se le pide hacer sus tablas de tiempos, pero sí razonar si los tiempos coinciden o no con la complejidad temporal de cada algoritmo.

Sustraccion1: a = 1; b = 1; k = 0; O(n)

Sustraccion2: a = 1; b = 1; k = 1; O(n^2)

Sustraccion3: a = 2; b = 1; k = 0; O(2^n)

Los tiempos obtenidos coinciden con la complejidad temporal de cada algoritmo ya que:

- Para Sustraccion1, según se duplica el tamaño del problema, el tiempo de ejecución también se ve duplicado.
- Para Sustraccion2, según se duplica el tamaño del problema, el tiempo de ejecución también se ve aumentado en un factor 2^2 = 4.
- Para Sustraccion3, según se incrementa en una unidad el tamaño del problema, el tiempo de ejecución se ve aumentado en un factor 2^1 = 2.

Contestar: ¿para qué valor de n dejan de dar tiempo (abortan) las clases Sustraccion1 y Sustraccion2? ¿por qué sucede eso?

Sustraccion1: n = 16000

Sustraccion2: n = 16000

Porque se desborda la pila al haber un exceso de llamadas recursivas en las que se tarda demasiado en llegar al caso de parada.

Calcular: ¿cuántos años tardaría en finalizar la ejecución Sustracción3 para n=80?

O(2^n)

n1 = 20 ; t1 = 78 msg

n2 = 80 ; ¿t2?

t2 = ((2^n2) / (2^n1)) * t1 = (lo pasamos a años…) = 2,8516*10^9 años

Implementar una clase Sustraccion4.java con una complejidad O(n^3) (con su pertinente toma de tiempos) y después rellenar una tabla en la que se muestre el tiempo (en msg.) para n=100, 200, 400, 800, … (hasta FdT).

a = 1; b = 1 (cualquiera porque no depende de este valor); k = 2

N	t (en msg)
100	5,6
200	43,3
400	343
800	2735
1600	21820
3200	FdT

Implementar una clase Sustraccion5.java con una complejidad O(3^(n/2)) (con su pertinente toma de tiempos) y después rellenar una tabla en la que se muestre el tiempo (en msg.) para n=30, 32, 34, 36, … (hasta FdT).

a = 3; b = 2; k = 0 (cualquiera porque no depende de este valor)

N	t (en msg)
30	627
32	1885
34	5637
36	16928
38	51270
40	FdT

Calcular: ¿cuántos años tardaría en finalizar la ejecución Sustracción5 para n=80?

O(3^(n/2))

n1 = 30 ; t1 = 627 msg

n2 = 80 ; ¿t2?

t2 = ((3 ^ (n2/2)) / (3 ^ (n1/2))) * t1 = (lo pasamos a años…) = 16,8458*10^3 años

Ejercicio 2:

Tras analizar la complejidad de las tres clases anteriores (Division1.java, Division2.java y Division3.java), no se le pide hacer sus tablas de tiempos, pero sí razonar si los tiempos coinciden o no la complejidad temporal de cada algoritmo.

Division1: a = 1; b = 3; k = 1; O(n)

Division2: a = 2; b = 2; k = 1; O(n*log(n))

Division3: a = 2; b = 2; k = 0; O(n^log2(2)) = O(n)

Los tiempos obtenidos coinciden con la complejidad temporal de cada algoritmo ya que:

- Para Division1, según se duplica el tamaño del problema, el tiempo de ejecución también se ve duplicado.
- Para Division2, según se duplica el tamaño del problema, el tiempo de ejecución se ve casi duplicado por ser una complejidad casi lineal.
- Para Division3, según se duplica el tamaño del problema, el tiempo de ejecución también se ve duplicado.

Implementar una clase Division4.java con una complejidad O(n^2) (con a<b^k) y la pertinente toma de tiempos. Después rellenar una tabla en la que se muestre el tiempo (en msg.) para n=1000, 2000, 4000, 8000, … (hasta FdT). [a = 1; b = 2; k = 2]

N	t (en msg)
1000	17,2
2000	63
4000	296
8000	1172
16000	4719
32000	19723
64000	FdT

Implementar una clase Division5.java con una complejidad O(n^2) (con a>b^k) y la pertinente toma de tiempos. Después rellenar una tabla en la que se muestre el tiempo (en msg.) para n=1000, 2000, 4000, 8000, … (hasta FdT). [a = 4; b = 2; k = 1]

N	t (en msg)
1000	45,0
2000	172
4000	718
8000	2781
16000	11281
32000	45719
64000	FdT

<u>Ejercicio 3:</u>

Tras analizar la complejidad de los diversos algoritmos que hay dentro de las dos clases (VectorSuma.java y Fibonacci.java), ejecutarlas y tras poner en una tabla los tiempos obtenidos, comparar la eficiencia de cada algoritmo.

VectorSuma:

Versión 1 (iterativa) [Metodo1]: O(n)

Versión 2 (recursiva con sustracción) [Metodo2]: a = 1; b = 1; k = 0; O(n)

Versión 3 (recursiva con división) [Metodo3]: a = 2; b = 2; k = 0; O(n)

	t (en msg)	**Eficiencia (comparado con…)**		
		Metodo1	**Metodo2**	**Metodo3**
Metodo1	0,000859	--	0,29712902	0,1970183
Metodo2	0,002891	3,36554133	--	0,6630734
Metodo3	0,00436	5,07566938	1,50812868	--

La comparación de eficiencia de cada versión del algoritmo se ha realizado dividiendo el tiempo obtenido en cada versión.

Observamos que el Metodo1 es el más rápido de los tres por tener el tiempo de ejecución más bajo y porque al dividir el tiempo que obtenemos con éste entre los tiempos obtenidos en Metodo2 y Metodo3 nos sale un resultado < 1, es decir, la afirmación se refuerza con dichos resultados.

El Metodo2 es más lento que Metodo1 (porque el resultado de dividir el tiempo obtenido en Metodo2 entre el tiempo obtenido en Metodo1 es > 1) pero más rápido que Metodo3 (porque el resultado de dividir el tiempo obtenido en Metodo2 entre el tiempo obtenido en Metodo3 es < 1).

Y el Metodo3 es el más lento de todos porque el resultado de dividir el tiempo obtenido entre el tiempo obtenido en el resto de métodos es > 1.

Fibonacci:

Versión 1 (iterativa) [Metodo1]: O(n)

Versión 2 (iterativa) [Metodo2]: O(n)

Versión 3 (recursiva con sustracción) [Metodo3]: a = 1; b = 1; k = 0; O(n)

Versión 4 (recursiva con sustracción) [Metodo4]: a = 2; b = 1 o b = 2; k = 0; O(1,6^n)

	t (en msg)	Eficiencia (comparado con…)			
		Metodo1	**Metodo2**	**Metodo3**	**Metodo4**
Metodo1	0,000375	--	0,685557587	0,39978678	6,1224E-08
Metodo2	0,000547	1,458666667	--	0,58315565	8,9306E-08
Metodo3	0,000938	2,501333333	1,714808044	--	1,5314E-07
Metodo4	6125	16333333,33	11197440,59	6529850,75	--

La comparación de eficiencia de cada versión del algoritmo se ha realizado dividiendo el tiempo obtenido en cada versión.

Observamos que el Metodo1 es el más rápido de los cuatro por tener el tiempo de ejecución más bajo y porque al dividir el tiempo que obtenemos con éste entre los tiempos obtenidos en Metodo2, Metodo3 y Metodo4 nos sale un resultado < 1, es decir, la afirmación se refuerza con dichos resultados.

El Metodo2 es más lento que Metodo1 (porque el resultado de dividir el tiempo obtenido en Metodo2 entre el tiempo obtenido en Metodo1 es > 1) pero más rápido que Metodo3 (porque el resultado de dividir el tiempo obtenido en Metodo2 entre el tiempo obtenido en Metodo3 es < 1) y que Metodo4 (porque el resultado de dividir el tiempo obtenido en Metodo2 entre el tiempo obtenido en Metodo4 es < 1).

El Metodo3 es más lento que Metodo1 (porque el resultado de dividir el tiempo obtenido en Metodo3 entre el tiempo obtenido en Metodo1 es > 1) y que Metodo2 (porque el resultado de dividir el tiempo obtenido en Metodo3 entre el tiempo obtenido en Metodo2 es > 1), pero es más rápido que Metodo4 (porque el resultado de dividir el tiempo obtenido en Metodo3 entre el tiempo obtenido en Metodo4 es < 1)

Y el Metodo4 es el más lento de todos porque el resultado de dividir el tiempo obtenido entre el tiempo obtenido en el resto de métodos es > 1.

Recordemos que si el resultado de una división:

- Es < 1 si el denominador es mayor que el numerador.
- Es > 1 si el numerador es mayor que el denominador.

Ejercicio 4:

Implementar una clase Mezcla.java que ordene los elementos de un vector.

Implementar una clase MezclaTiempos.java que tras ser ejecutada sirva para rellenar la siguiente tabla (análoga a las vistas en la práctica 2):

TABLA ALGORITMO MEZCLA (tiempos en milisegundos y SIN_OPTIMIZACIÓN):

Pondremos "FdT" para tiempos superiores al minuto y "FdF" los menores a 50 msg

N	t ordenado	t inverso	t aleatorio
31250	FdF	FdF	FdF
62500	FdF	FdF	FdF
125000	82	81	87
250000	169	169	182
500000	354	352	385
1000000	738	735	809
2000000	1558	1538	1681
4000000	3209	3191	3484
8000000	6646	6623	7270
16000000	13799	13748	15089
32000000	28733	28633	31331
64000000	59557	59088	FdT
128000000	FdT	FdT	FdT

Rellene la tabla siguiente, trasladando a ella los tiempos obtenidos para el Rápido en el caso aleatorio en la práctica 2 y los obtenidos aquí para el Mezcla en el caso aleatorio. ¿Qué constante le sale como comparación de ambos algoritmos?

TABLA ALGORITMO MEZCLA vs. RÁPIDO (caso aleatorio) (tiempos en milisegundos y SIN_OPTIMIZACIÓN):

Pondremos "FdT" para tiempos superiores al minuto y "FdF" los menores a 50 msg

N	t mezcla(t1)	t rapido(t2)	t1/t2
250000	187	187	1
500000	385	395	0,974683544
1000000	800	901	0,887902331
2000000	1675	2219	0,754844525
4000000	3479	5901	0,589561091
8000000	7217	18082	0,399126203
16000000	15008	FdT	#¡VALOR!

Ambos son O(n*log(n))

La constante obtenida, la cual se encuentra en la columna de t1/t2, no tiene un valor constante para cualquier tamaño de problema n, ya que varía de un tamaño a otro.

Sin embargo, podemos observar que el valor de la constante en cada tamaño de problema n es <= 1, es decir, que el numerador es menor que el denominador. Luego, esto refuerza la afirmación de que los tiempos de mezcla comparados con los de rápido son menores pese a que tengan ambos algoritmos la misma complejidad.