Ejercicio 1:

Tras medir los tiempos, rellenar la tabla:

TABLA 1 (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
10000	641	1484	1172
2*10000	2562	5968	4624
22*10000**	10247	23776	18543
23*10000**	41023	FdT	FdT
24*10000**	FdT	FdT	FdT

Razone si los diferentes tiempos obtenidos concuerdan con lo esperado, según la complejidad temporal estudiada.

Los resultados obtenidos concuerdan con lo esperado ya que la complejidad temporal del algoritmo de la burbuja es de $O(n^2)$ en las tres versiones del vector a ordenar.

Sin embargo, los tiempos obtenidos no son iguales en todos los casos en los que se presenta el vector a ordenar, es decir, para un vector ordenado no se van a ejecutar las instrucciones del condicional interno nunca, luego, obtenemos un tiempo menor que en los otros casos.

Para un vector inversamente ordenado se ejecutan siempre las instrucciones del condicional interno, dando lugar al tiempo de ejecución más alto de los tres casos.

Para un vector aleatoriamente ordenado no se ejecutan siempre las instrucciones del condicional pero en algunas iteraciones si llegan a ser ejecutadas, dando lugar a un "punto medio" de los tiempos de ejecución entre los tres casos.

Ejercicio 2:

Implementar una clase SeleccionTiempos.java que tras ser ejecutada sirva para rellenar la siguiente tabla:

TABLA 2 (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
10000	563	625	563
2*10000	2234	2500	2250
22*10000**	8936	9937	8953
23*10000**	35759	39778	35826
24*10000**	FdT	FdT	FdT

Razone si los diferentes tiempos obtenidos concuerdan con lo esperado, según la complejidad temporal estudiada.

Los resultados obtenidos concuerdan con lo esperado ya que la complejidad temporal del algoritmo de selección es de $O(n^2)$ en las tres versiones del vector a ordenar.

Sin embargo, los tiempos obtenidos no son iguales en todos los casos en los que se presenta el vector a ordenar, es decir, para un vector ordenado no se van a ejecutar las instrucciones del condicional interno nunca, luego, obtenemos un tiempo menor que en los otros casos.

Para un vector inversamente ordenado se ejecutan siempre las instrucciones del condicional interno, dando lugar al tiempo de ejecución más alto de los tres casos.

Para un vector aleatoriamente ordenado no se ejecutan siempre las instrucciones del condicional pero en algunas iteraciones si llegan a ser ejecutadas, dando lugar a un "punto medio" de los tiempos de ejecución entre los tres casos.

Ejercicio 3:

Implementar una clase InsercionTiempos.java que tras ser ejecutada sirva para rellenar la siguiente tabla:

TABLA 3 (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
10000	FdF	985	484
2*10000	FdF	3828	1921
22*10000**	FdF	15310	7640
23*10000**	FdF	FdT	30591
24*10000**	FdF	FdT	FdT
25*10000**	FdF	FdT	FdT
26*10000**	FdF	FdT	FdT
27*10000**	FdF	FdT	FdT
28*10000**	63	FdT	FdT
29*10000**	125	FdT	FdT
210*10000**	265	FdT	FdT
211*10000**	516	FdT	FdT
212*10000**	1047	FdT	FdT
213*10000**	2109	FdT	FdT

Razone si los diferentes tiempos obtenidos concuerdan con lo esperado, según la complejidad temporal estudiada.

Los resultados obtenidos concuerdan con lo esperado ya que la complejidad temporal del algoritmo de inserción es de O(n) para un vector inicialmente ordenado, y $O(n^2)$ para el resto de versiones del vector a ordenar.

Como para un vector inicialmente ordenado la complejidad es lineal, podemos observar en los tiempos recogidos para este caso que los tiempos según se duplica el tamaño del problema, los tiempos obtenidos se multiplican por un factor de 2.

Para un vector ordenado inversamente obtenemos los tiempos de ejecución más altos de los tres casos porque siempre va a ejecutar el bucle while interno, dando lugar al tiempo de ejecución más alto posible.

Para un vector ordenado aleatoriamente obtenemos un tiempo menor que para el inversamente ordenado porque no siempre se va a ejecutar el bucle while interno, luego, el tiempo de ejecución para este caso con respecto al caso en el que el vector está inversamente ordenado son menores y mayores que los del caso en el que el vector está inicialmente ordenado porque la complejidad en el aleatorio es cuadrática y en el ordenado es lineal.

Ejercicio 4:

Implementar una clase RapidoTiempos.java que tras ser ejecutada sirva para rellenar la siguiente tabla:

TABLA 4 (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
250000	125	140	188
2*250000	234	281	375
22*250000**	516	562	890
23*250000**	1063	1172	2219
24*250000**	2219	2437	5984
25*250000**	4640	5093	18272
26*250000**	9624	10562	FdT

Razone si los diferentes tiempos obtenidos concuerdan con lo esperado, según la complejidad temporal estudiada.

Los resultados obtenidos concuerdan con lo esperado ya que la complejidad temporal del algoritmo rápido es de O(n*log(n)) en las tres versiones del vector a ordenar.

Sin embargo, los tiempos obtenidos no son iguales en todos los casos en los que se presenta el vector a ordenar, es decir, para un vector ordenado se van a realizar menos intercambios de los elementos de los subvectores que en el caso en el que el vector está ordenado inversamente.

A su vez, en el caso en el que el vector está ordenado inversamente se realizan menos operaciones de intercambio de los elementos de los subvectores que en el caso en el que el vector está aleatoriamente ordenado inicialmente, dando lugar a un "punto medio" de los tiempos de ejecución entre los tres casos en los que se puede presentar el vector a ordenar.

Tras ver lo que tarda el rápido en ordenar 16 millones de elementos inicialmente en orden aleatorio, calcule (a partir de las complejidades y los datos de las tablas anteriores), ¿cuántos días tardaría cada uno de esos tres métodos (Burbuja, Selección e Inserción) en hacer lo mismo?

- Burbuja (Complejidad $O(n^2)$):

n1 = 10.000 ; t1 = 1172 msg

n2 = 16.000.000 ; ¿t2?

t2 = (n2^2 / n1^2) * t1 = … = 34,72 dias

- Selección (Complejidad $O(n^2)$):

n1 = 10.000 ; t1 = 563 msg

n2 = 16.000.000 ; ¿t2?

t2 = (n2^2 / n1^2) * t1 = … = 16,68 dias

- Inserción (Complejidad $O(n^2)$):

n1 = 10.000 ; t1 = 484 msg

n2 = 16.000.000 ; ¿t2?

t2 = (n2^2 / n1^2) * t1 = … = 14,34 dias

Ejercicio 5:

Ahora se propone hacer una toma de tiempos, que aunque no sea concluyente y determinante para compararlos, sí será orientativa. Para ello, implementar una clase RapidoInsercionTiempos.java que tomará tiempos (para un tamaño de n=16 millones de elementos generados inicialmente de forma aleatoria) para rellenar la siguiente tabla:

TABLA 5 (tiempos en milisegundos y SIN_OPTIMIZACIÓN):

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

Caso	t aleatorio
Rápido	49178
Rápido+Inserción (k=5)	FdT
Rápido+Inserción (k=10)	48131
Rápido+Inserción (k=20)	47355
Rápido+Inserción (k=30)	46290
Rápido+Inserción (k=50)	43516
Rápido+Inserción (k=100)	32217
Rápido+Inserción (k=200)	11440
Rápido+Inserción (k=500)	19778
Rápido+Inserción (k=1000)	39857

Razone conclusiones obtenidas a partir de la tabla anterior.

Observamos que al incrementar el valor de k para que las particiones de dicho tamaño sean ordenadas por el método de inserción, el tiempo de ejecución disminuye hasta que el valor de k toman valores suficientemente altos como para que el algoritmo de inserción nos dé un tiempo de ejecución mayor debido a que se está trabajando con particiones de un tamaño muy grande y el algoritmo de inserción para un vector ordenado aleatoriamente es de complejidad cuadrática, lo que castiga mucho a la eficiencia del algoritmo RAPIDO+INSERCION.