Prueba No:	1.
Tipo de Prueba:	Unitaria Automática
Técnica:	JUnit, con casos de prueba implementados como atributos dentro de las clases de pruebas
Método a Probar:	fillListWithNumbersInAInterval (char numberType, char cloneNumber, Object[] list, int startInterval,int endInterval, Random rand)
Descripción de Entradas:	Este método recibe como parámetros el tipo de elemento que el usuario le gustaría generar, ya sean enteros o floats. Tambien, recibe como parámetro otro char, cloneNumber, el cual dice si los elementos se podrán generar repetidos o no. Recibe una lista vacía, en donde se generarán los elementos aleatorios, un entero del limite inferior y superior, y un Random que me genera los numeros aleatoriamente.
Descripción de Salidas:	No me regresa nada como salida, es de tipo void.

Caso de Prueba No:	1.1. testFillListWithNumbersInAInterval()
Objetivo:	Quiero saber si con el método testFillListWithNumbersInAInterval() me genera aleatoriamente una lista de elementos tanto de coma flotante como de enteros
Escenario:	setupScenario1(); Creamos dos objetos tipo lista, de tamaño 4 cada uno y una variable Random.
Entrada(s):	Este método recibe como parámetro un chart el cual me indica si el usuario escogió Enteros o de coma flotante para ser generados. Por otra parte, recibe como parámetro otro chart el cual me va a indicar si los elementos a generar pueden repetir numeros. Recibe dos enteros los cuales son el límite inferior y superior de los elementos a generar. Este método recibe como parámetro una lista vacía de tamaño 4 y una variable de tipo.
Salida Esperada:	La salida esperada es que me genere una lista ya sea de enteros o de coma flotante.

Prueba No:	2. QuickSort
Tipo de Prueba:	Unitaria Automática
Técnica:	JUnit, con casos de prueba implementados como atributos dentro de las clases de pruebas
Método a Probar:	**quickSort**(**int**[] list, **int** high, **int** low)
Descripción de Entradas:	Este método recibe como parámetros una lista de enteros, la cual va aordenar, tambien recibe como parámetro un entero High el cual le indicara al usuario el tamaño de la lista y otro entero Low, el cual le indicara al usuario desde donde empezara el arreglo, en este caso desde cero.
Descripción de Salidas:	No me regresa nada como salida, es de tipo void.

Caso de Prueba No:	2.1. testQuickSort()
Objetivo:	Quiero saber si con el método **testQuickSort()** me ordena ascendentemente una lista de elementos tanto de coma flotante como de enteros
Escenario:	setupScenario1(); Creamos dos objetos tipo lista, de tamaño 4 cada uno y una variable Random.

Entrada(s):	Este método no recibe como parámetro nada, pero el método que evaluaremos dentro del sí. Este método recibe como parámetros una lista enteros, la cual va a ordenar, tambien recibe como parámetro un entero High el cual le indicara al usuario el tamaño de la lista y otro entero Low, el cual le indicara al usuario desde donde empezara el arreglo, en este caso desde cero. Yo cree 6 listas de diferentes tamaños y tipos. Las puse en pares, una es la lista que quería ordenar y la otra era la lista ya ordenada y comparaba si las dos listas eran iguales.
Salida Esperada:	La salida esperada es que las dos listas creadas sean iguales. Tanto la que no estaba ordenada, como la que esta ordenada, despues de haber ejecutado el método quickSort.

Prueba No:	3. pigeonHoleSort()
Tipo de Prueba:	Unitaria Automática
Técnica:	JUnit, con casos de prueba implementados como atributos dentro de las clases de pruebas
Método a Probar:	pigeonHoleSort(int[] list)
Descripción de Entradas:	Este método recibe como parámetros una lista de enteros, la cual va a ordenar. Una vez recibida lla lista el método ejecuta sus comandos y procede a organizarla.
Descripción de Salidas:	No me regresa nada como salida, es de tipo void.

Caso de Prueba No:	3.1. testpigeonHoleSort()
Objetivo:	Quiero saber si con el método **testpigeonHoleSort()** me ordena ascendentemente una lista de elementos de enteros
Escenario:	**setupScenario3();** en este escenario creamos dos listas de tamaño 6, la primera es la lista que se quiere ordenar y la segunda es la misma lista , pero ya ordenada que es con la cual vamos a comparar. **setupScenario4();** en este escenario creamos dos listas de tamaño 11, la primera es la lista que se quiere ordenar y la segunda es la misma lista , pero ya ordenada que es con la cual vamos a comparar. **setupScenario5();** en este escenario creamos dos listas de tamaño 21, la primera es la lista que se quiere ordenar y la segunda es la misma lista , pero ya ordenada que es con la cual vamos a comparar.
Entrada(s):	Este método no recibe como parámetro nada, pero el método que evaluaremos dentro del sí. Este método recibe como parámetros una lista de enteros, la cuales es la que vamos a ordenar.
Salida Esperada:	La salida esperada es que las dos listas creadas sean iguales. Tanto la que no estaba ordenada, como la que esta ordenada, despues de haber ejecutado el método PigeonSort().

Prueba No:	4. RadixSort()
Tipo de Prueba:	Unitaria Automática
Técnica:	JUnit, con casos de prueba implementados como atributos dentro de las clases de pruebas
Método a Probar:	pigeonHoleSort(int[] list)
Descripción de Entradas:	Este método recibe como parámetros una lista de floats, la cual va a ordenar. Una vez recibida la lista el método ejecuta sus comandos y procede a organizarla.
Descripción de Salidas:	No me regresa nada como salida, es de tipo void.

Caso de Prueba No:	3.1. testRadixSort()
Objetivo:	Quiero saber si con el método **testRadixSort()** me ordena ascendentemente una lista de elementos de enteros
Escenario:	**setupScenario6();** en este escenario creamos dos listas de tamaño 10, la primera es la lista que se quiere ordenar y la segunda es la misma lista , pero ya ordenada que es con la cual vamos a comparar. **setupScenario7();** en este escenario creamos dos listas de tamaño 20, la primera es la lista que se quiere ordenar y la segunda es la misma lista , pero ya ordenada que es con la cual vamos a comparar. **setupScenario8();** en este escenario creamos dos listas de tamaño 21, la primera es la lista que se quiere ordenar y la segunda es la misma lista , pero ya ordenada que es con la cual vamos a comparar. **setupScenario9();** en este escenario creamos dos listas de tamaño 30, la primera es la lista que se quiere ordenar y la segunda es la misma lista , pero ya ordenada que es con la cual vamos a comparar.
Entrada(s):	Este método no recibe como parámetro nada, pero el método que evaluaremos dentro del sí. Este método recibe como parámetros una lista de floats, la cuales es la que vamos a ordenar.
Salida Esperada:	La salida esperada es que las dos listas creadas sean iguales. Tanto la que no estaba ordenada, como la que esta ordenada, despues de haber ejecutado el método RadixSort().

Prueba No:	5. ReadnumbersFile(File file, char numberType)
Tipo de Prueba:	Unitaria Automática
Técnica:	JUnit, con casos de prueba implementados como atributos dentro de las clases de pruebas
Método a Probar:	ReadnumbersFile(File file, char numberType)
Descripción de Entradas:	Este método recibe como parámetros una archivo plano, el cual va a convertirlo para poderlo leer en el sistema, ya sea un archivo plano de enteros o de Floats..
Descripción de Salidas:	No me regresa nada como salida, es de tipo void.

Caso de Prueba No:	5.1. testReadNumbersFile01()
Objetivo:	Quiero saber si con el método **testReadNumbersFile01()** me va a leer el archivo plano para poder ingresarlo al sistema y asi poder ordenar los elementos.
Escenario:	**En ese escenario creamos un archivo plano, el cual pasaremos como parámetro en el método que vamos a evaluar**
Entrada(s):	Este método no recibe como parámetro nada, pero el método que evaluaremos dentro del sí. Este método recibe como parámetros una lista de floats, la cuales es la que vamos a ordenar.
Salida Esperada:	Este método no retorna nada, pero me ingresa el archivo plano al sistemas.