Departamento de TIC
Algoritmos y Estructuras de Datos
Unidad 1 Laboratorio

Integrantes: Nicolás Martinez, Cristian Molina y Juan Manuel Castillo.

GitHub url: https://github.com/nicolasmr21/SortSoftware

Paso 1. Identificación del problema

Identificación de necesidades y síntomas

- La empresa de fabricación de microprocesadores requiere resolver operaciones de aritmética, procesamiento gráfico, procesamiento de señales, procesado de criptografía, entre otros.
- La solución debe estar implementada en tres algoritmos de ordenamiento que permitan distribuir organizadamente números enteros de gran tamaño y números de coma flotante de cualquier dimensión.
- La empresa necesita el desarrollo de un microprocesador que libere cargas al procesador principal (CPU).
- La solución debe estar implementada en software el cual funcionará como prototipo de pruebas para operaciones nativas en hardware.

Definición del problema

Una empresa de fabricación de microprocesadores está evaluando la posibilidad de desarrollar varios algoritmos de ordenamiento como instrucciones básicas de su próximo coprocesador matemático para la disminución de carga del procesador principal (CPU), acelerando el rendimiento del sistema.

Requerimientos funcionales

Nombre	**R.F. # 1 Permitir al usuario el ingreso de valores.**
Resumen	**Este requerimiento se encarga de permitirle ingresar valores tanto enteros como de coma flotante al usuario.**
Entrada	**Valores.**
Salida	**Se han ingresado los valores a ordenar por medio de la interfaz.**

Nombre	R.F. # 2 Generar aleatoriamente valores.
Resumen	Este requerimiento se encarga de generar aleatoriamente valores tanto enteros como de coma flotante.
Entrada	Restricciones.
Salida	Se ha generado los valores enteros y de coma flotante mediante la interfaz.

Nombre	R.F. # 3 Configurar la cantidad total de valores a generar mediante un intervalo.
Resumen	Este requerimiento se encarga de configurar la cantidad total de números a generar y el intervalo en el cual se generarán los números.
Entrada	Cantidad de valores requeridos.
Salida	Se ha configurado la cantidad total de números en un intervalo.

Nombre	R.F. # 4 Permitir indicar si los valores generados pueden repetirse o ser diferentes.
Resumen	Este requerimiento se encarga de indicar si los números generados tienen la posibilidad de repetirse o ser todos diferentes.
Entrada	Verificación.
Salida	Se ha indicado si los valores pueden repetirse o son diferentes.

Nombre	R.F. # 5 Generar los valores con orden aleatorio
Resumen	Este requerimiento se encarga de generar los valores en un orden aleatorio.
Entrada	-Verificación.
Salida	Se han generado los valores aleatoriamente.

Nombre	R.F. # 6 Generar los valores de manera inversa.
Resumen	Este requerimiento se encarga de generar los valores de forma inversa.
Entrada	-Verificación.
Salida	Se han generado inversamente los valores de forma ordenada

Nombre	R.F. # 7 Generar los valores ya ordenados.
Resumen	Este requerimiento se encarga de generar los valores de forma ordenada.
Entrada	-Verificación.
Salida	Se han generado los valores de forma ordenada.

Nombre	R.F. # 8 Generar los valores de acuerdo a un porcentaje de desorden dado por el usuario.
Resumen	Este requerimiento permite que el usuario pueda definir el nivel de desordenamiento de los datos mediante un porcentaje para así ser generados
Entrada	-Verificación.
Salida	Se han generado los valores de acuerdo al porcentaje de desorden indicado por el usuario.

Nombre	R.F. # 9 Ordenar los valores generados utilizando el algoritmo apropiado de los tres elegidos.
Resumen	Este requerimiento permite que se ordenen los valores generados usando el algoritmo que sea más pertinente.
Entrada	Valores.
Salida	Se ha ordenado los valores usando el algoritmo más apropiado.

Nombre	R.F. # 10 Restringir o permitir el algoritmo de ordenamiento a usar dependiendo de la cantidad e intervalo de los datos.
Resumen	Este requerimiento se encarga de restringir o permitir el uso de un algoritmo que sea más óptimo para la cantidad e intervalo de datos dada.
Entrada	Valores.
Salida	Se ha restringido/permitido X algoritmo para la cantidad de datos e intervalo a usar.

Nombre	R.F. # 11 Presentar el tiempo que toma el algoritmo de ordenamiento.
Resumen	Este requerimiento se encarga de informar cuanto tiempo se toma un algoritmo de ordenamiento en realizar su función.
Entrada	Valores.
Salida	Se ha presentado el tiempo que tarda el algoritmo de ordenamiento.

Nombre	R.F. # 12 Generar ordenada la secuencia. (Con base al porcentaje de desorden)
Resumen	Este requerimiento se encarga de generar la secuencia de forma ordenada con base al porcentaje de desorden.
Entrada	Valores.
Salida	Se ha generado ordenadamente la secuencia.

Nombre	R.F. # 13 Obtener un número K de cuantas posiciones deben estar desordenadas. (Con base al porcentaje de desorden y tamaño de secuencia)
Resumen	Este requerimiento se encarga de obtener un número K con base al porcentaje de desorden y el tamaño de secuencia.
Entrada	Valores.
Salida	Se ha obtenido un número K de las posiciones que deben estar desordenadas.

Nombre	R.F. # 14 Generar K/2 pares posiciones diferentes (Con base al porcentaje de desorden)
Resumen	Este requerimiento se encarga de generar K/2 pares posiciones diferentes con base al porcentaje de desorden.
Entrada	Valores.
Salida	Se ha generado K/2 pares posiciones diferentes.

Nombre	R.F. # 15 Intercambiar los valores entre cada K/2 posición par (Con base al porcentaje de desorden y las K/2 pares posiciones)
Resumen	Este requerimiento se encarga de intercambiar los valores entre cada K/2 par con base al porcentaje de desorden y las K/2 pares posiciones.
Entrada	Valores.
Salida	Se han intercambiado los valores entre cada K/2 posición par.

Requerimientos no funcionales

Nombre	R.N.F# 1 Limitar la organización de valores a 3 algoritmos de ordenamiento
Resumen	Los valores ingresados ó generados sólo podrán ser ordenados mediante 3 algoritmos de los cuales se elegirá el mejor dependiendo del tipo o cantidad de datos.

Fase 2. Recopilación de la información necesaria (JUANMA)

Definiciones

Fuente:

https://www.inf.utfsm.cl/

https://es.wikipedia.org/

https://www.softwaredoit.es/

https://es.quora.com/

http://www.codexion.com/

Coprocesador

Es un microprocesador de un ordenador utilizado como suplemento de las funciones del procesador principal (la CPU). Las operaciones ejecutadas por uno de estos coprocesadores pueden ser operaciones de aritmética, procesamiento gráfico, de señales, procesado de texto o criptografía, etcétera. (Fuente: Wikipedia.)

Funcionamiento de un Coprocesador

Se da a través de distintos pasos que combinan instrucciones almacenadas en código binario. El sistema lee la instrucción desde la memoria, la envía al decodificador que determinará de qué se trata y cuáles son los pasos a seguir. Finalmente se ejecuta la instrucción y los resultados estarán recopilados en la memoria o en los registros. Hay diversos tipos de procesadores cada uno con característica y capacidades diversas de acuerdo con los intereses del usuario. (Fuente: SoftwareDoit.)

Coma Flotante

Es una forma de notación científica usada en los microprocesadores con la cual se pueden representar números racionales extremadamente grandes y pequeños de una manera muy eficiente y compacta, y con la que se pueden realizar operaciones aritméticas. (Fuente: Wikipedia.)

Algoritmo de Ordenamiento

Consiste en poner elementos de una lista o un vector en una secuencia dada por una relación de orden, es decir, el resultado de salida ha de ser una permutación —o reordenamiento de la entrada que satisfaga la relación de orden dada. Las relaciones de orden más usadas son el orden numérico y el orden lexicográfico**.** (Fuente: Wikipedia.)

Necesidad de los algoritmos de ordenamiento

Son importantes para optimizar el uso de otros algoritmos (como los de búsqueda y fusión) que requieren listas ordenadas para una ejecución rápida. También es útil para poner datos en forma canónica y para generar resultados legibles por humanos. (Fuente: Wikipedia.)

Clasificación de algoritmos de ordenamiento

Complejidad computacional

Trata de clasificar los problemas que pueden, o no pueden ser resueltos con una cantidad determinada de recursos. A su vez, la imposición de restricciones sobre estos recursos es lo que la distingue de la teoría de la computabilidad, la cual se preocupa por qué tipo de problemas pueden ser resueltos de manera algorítmica. (Fuente: Wikipedia.)

Algoritmo In-place

Es un algoritmo que transforma la entrada sin estructura de datos auxiliares. Sin embargo, se permite una pequeña cantidad de espacio de almacenamiento adicional para las variables auxiliares. La entrada suele ser sobrescrita por la salida a medida que se ejecuta el algoritmo. El algoritmo in place no utiliza una estructura de datos auxiliar para realizar la ordenación. En otras palabras: no utiliza memoria adicional innecesariamente. (Fuente: Quora, Alan Chavez)

Eficiencia Algorítmica

Con el objetivo de lograr una eficiencia máxima se quiere minimizar el uso de recursos. Sin embargo, varias medidas (ej. complejidad temporal, complejidad espacial) no pueden ser comparadas directamente, luego, cuál de dos algoritmos es considerado más eficiente, depende de cual medida de eficiencia se está considerando como prioridad, ej. la prioridad podría ser obtener la salida del algoritmo lo más rápido posible, o que minimice el uso de la memoria, o alguna otra medida particular. (Fuente: Wikipedia.)

Dato primitivo

El tipo de dato de una variable determina los valores que puede contener además de las operaciones que se puede realizar sobre ella. El lenguaje de programación Java incluye siete otros tipos de datos primitivos además de int. Un tipo primitivo está predefinido por el lenguaje y se nombra con una palabra clave reservada. Los valores primitivos no comparten estado con otros valores primitivos. Los ocho tipos de datos primitivos incluidos en el lenguaje de programación Java son: Byte de 8 bits, short de 16 bits, int de 32 bits, long de 64 bits, float de 32 bits, double de 63 bits, boolean de falso y verdadero, char de 16 bits. (Fuente: Codexion.)

Fase 3. Búsqueda de soluciones creativas

ALTERNATIVAS DE ORDENAMIENTO

Quicksort: Es un algoritmo que toma un “x” de una posición cualquiera del arreglo y trata de ubicar a “x” en la posición correcta del arreglo tal que todos los elementos a la izquierda del elemento sean menores o iguales que “x” y a su derecha mayores. Es de propósito general.

Mergesort: A través de la técnica de divide y conquistarás este algoritmo usa la recursividad para dividir el arreglo en dos partes haciendo que cada vez sea más pequeño hasta llegar a una unidad, ordena tal unidad y luego combina lo ordenado. Es de propósito general.

Heapsort: Este algoritmo consiste en generar un árbol binario amontonado en el cual se podrán ir extrayendo sus raíces para obtener un conjunto ordenado. Es de propósito general.

Radix: Este algoritmo ordena enteros procesando cada dígito del número de forma individual. Aunque no es solo para enteros, utiliza los dígitos para crear una jerarquía de importancia para así ordenar los valores. Es de propósito específico,

Counting: Este es un algoritmos para ordenar solo elementos contables como los enteros en el cual se determina el número de elementos de la misma clase para luego ser ordenados. debe conocer el intervalo en el que estan los numeros para luego así crear un vector en el cual se van a introducir los números de cada clase. Es de propósito específico.

Bucket: Distribuye todos los elementos en un número finito de casilleros, En cada uno de estos casillero se distribuirán los elementos, tales deben de cumplir unas restricciones para entrar en un casillero específico.

Binary tree sort: Ordena sus elementos haciendo uso de un árbol binario de búsqueda. Se basa en ir construyendo poco a poco el árbol binario introduciendo cada uno de los elementos, los cuales quedarán ya ordenados. Después, se obtiene la lista de los elementos ordenados recorriendo el árbol en *inorden*. (Fuente: wikipedia) Propósito general.

Introsort: Híbrido entre quicksort, heapsort e inserción.

ALTERNATIVAS DE GENERACIÓN ALEATORIA.

En *Java* existen dos clases principales para generar números aleatorios:

- java.util.Random
- java.security.SecureRandom

La función Math.random() usa java.util.Random por si acaso.

Mientras tanto, no es de gran importancia si los datos producidos son realmente aleatorios, Math.random() o Random hagan el trabajo, pero hay un problema:

Si se genera datos aleatorios con la misma secuencia de código y con la misma semilla, siempre se van a generar los mismos datos que parecen aleatorios, pero en realidad son reproducibles. Por la misma razón, típicamente se usa un sello de tiempo (como System.currentTimeMillis()) para generar nuevos datos. (Fuente: stackoverflow, usuario Stefan Nolde)

Clase java.util.Random

La clase java.util.Random debemos instanciarla, a diferencia del método Math.random().
A cambio, tendremos bastantes más posibilidades.

Fase 4. Transición de la formulación de ideas a los diseños preliminares

La revisión cuidadosa de las otras alternativas nos conduce a lo siguiente:

Ideas descartadas:

- **Binary tree sort y Heap:** Tales algoritmos necesitan que se implementara árboles binarios de búsqueda y quitaria la simplicidad de la solución.
- **Introsort:** poca información para su implementación.

Diseños preliminares

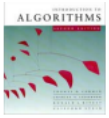
Quicksort

- *Requiere de pocos recursos en comparación a otros métodos de ordenamiento.*
- No se requiere de espacio adicional durante ejecución (in-place processing)
- Para ordenar una lista de números/nombres**.**

Pseudocode for quicksort

```
QUICKSORT(A, p, r)
  if p < r
    then q ← PARTITION(A, p, r)
      QUICKSORT(A, p, q–1)
      QUICKSORT(A, q+1, r)

Initial call: QUICKSORT(A, 1, n)
```

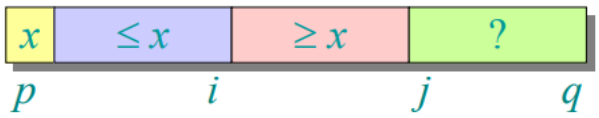



Partitioning subroutine

```
PARTITION(A, p, q)  ▷ A[p . . q]
  x ← A[p]          ▷ pivot = A[p]
  i ← p
  for j ← p + 1 to q
    do if A[j] ≤ x
         then  i ← i + 1
               exchange A[i] ↔ A[j]
  exchange A[p] ↔ A[i]
  return i
```

Running time = $O(n)$ for n elements.

Invariant:



Counting

- *Sólo ordena números enteros, no vale para ordenar cadenas y es desaconsejable para ordenar números decimales.*

Counting sort

```
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to n
    do C[A[j]] ← C[A[j]] + 1    ▷ C[i] = |{key = i}|
for i ← 2 to k
    do C[i] ← C[i] + C[i–1]     ▷ C[i] = |{key ≤ i}|
for j ← n downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] – 1
```


Bucket

- Es estable, cuando existen claves iguales se preserva el orden existente
- Este algoritmo es eficiente cuando la cantidad de casilleros es menor a la cantidad de claves. El tiempo para clasificar los elementos es constante, las claves repetidas ingresan en un mismo casillero, no se hace comparaciones entre claves.

Analysis of Bucket Sort

- Bucket-Sort(A)

1. Let B[0....n-1] be a new array
2. n = length[A]
3. for i = 0 to n-1
4. make B[i] an empty list
5. for i = 1 to n
6. do insert A[i] into list B[floor of n A[i]]
7. for i = 0 to n-1
8. do sort list B[i] with Insertion-Sort
9. Concatenate lists B[0], B[1],...,B[n-1] together in order

Step 5 and 6 takes O(n) time

Step 7 and 8 takes O(n log(n/k) time

Step 9 takes O(k) time

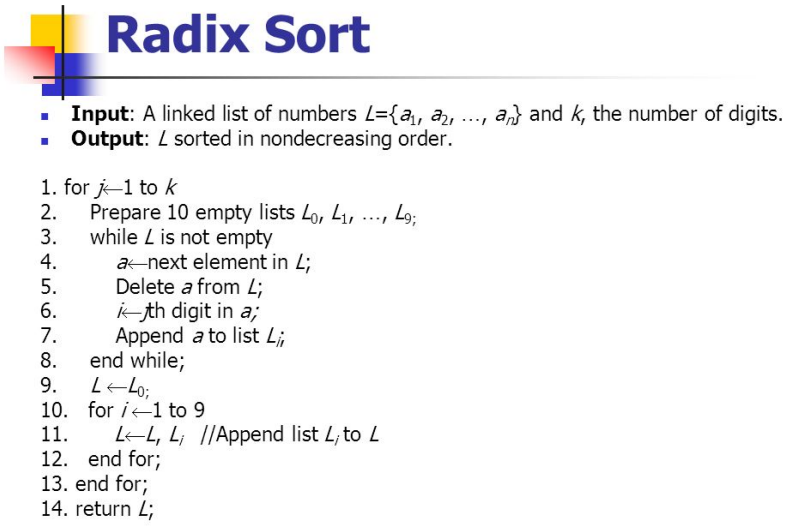
In total Bucket sort takes : O(n) (if k=Θ(n))

Fuente: Hossain Md Shakhawat Ph.D Student at Tokyo Institute of Technology

Radix

- Radix Sort es estable, preservando la orden de elementos iguales.

- El tiempo de ordenar cada elemento es constante, ya que no se hacen comparaciones entre elementos.
- Radix Sort no funciona tan bien cuando los números son muy largos, ya que el total de tiempo es proporcional a la longitud del número más grande y al número de elementos a ordenar.



***Fuente:* Kathleen Brooks**

Merge sort:

- Método de ordenamiento estable mientras la función de mezcla sea implementada correctamente.
- Muy estable cuando la cantidad de registros a acomodar es de índice bajo, en caso contrario gasta el doble del espacio que ocupan inicialmente los datos.
- está definido recursivamente. Si se deseara implementarla no recursivamente se tendría que emplear una pila y se requeriría un espacio adicional de memoria para almacenarla.

```
MERGE(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
 4  for i ← 1 to n1
 5       do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7       do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13       do if L[i] ≤ R[j]
14            then A[k] ← L[i]
15                 i ← i + 1
16            else A[k] ← R[j]
17                 j ← j + 1
```

Fuente: Thomas H. Cormen

Fase 5. Evaluación y selección de la mejor solución

Los algoritmos que se escogieron como mejores soluciones fueron: Quicksort, Radixsort y Countingsort.

Los criterios que se tuvieron en cuenta para evaluar los algoritmos fueron:

- Complejidad temporal (peor y mejor caso). 25% cada uno.
- Complejidad espacial (peor y mejor caso). 25% cada uno.
- Estabilidad. 5%
- Complejidad espacial auxiliar. 5%

Al evaluar las posibles soluciones, los algoritmos con mejor puntaje fueron: (ordenados de mayor a menor)

1. Radixsort.
2. Counting Sort.
3. Bucketsort.
4. Quicksort.
5. Mergesort.

Sin embargo, los tres algoritmos con mejor puntaje eran no comparativos. Por esto, se descartó el Bucketsort y se tomó el siguiente (Quicksort).

La rúbrica es la siguiente:

Algoritmo	Complejidad T (Peor caso)	Complejidad E (Peor caso)	Complejidad T (mejor caso)	Complejidad E (mejor caso)	Estabilidad	Complejidad E. Auxiliar	Puntaje
	25%	25%	20%	20%	5%	5%	100%
Quicksort	1	4	2	4	4	5	2,9
Mergesort	2	3	3	3	5	3	2,85
Radixsort	4	4	4	4	5	5	4,1
Countingsort	4	4	4	4	5	5	4,1
Bucketsort	4	4	4	4	5	5	4,1

O(n^2)	1
O(n*log(n))	2
O(n)	3
O(log(n))	4
O(1)	5
Estable	5
Inestable	4

A la hora de decidir qué algoritmo se va a usar para ordenar el arreglo, se observan las siguientes condiciones:

- La distancia entre el menor y el mayor número. (Peor caso del Counting).
- El número de dígitos del menor número. (Peor caso del Radix)
- El porcentaje de orden que tiene el arreglo. (Peor caso del Quick).

Estas son las soluciones para las posible combinaciones:

Floor > 100	10% desordenado	roof-floor > 100	
Números grandes	Casi ordenado	Intervalo grande	Solución
X	X		Counting
	X	X	Radix
X		X	Quick
X			Counting
		X	Radix
	X		Radix
X	X	X	Radix
			Radix

Fase 6. Preparación de informes y especificaciones. Diseño de diagrama de clases

- Diagrama de clases
- Pseudocódigo de algoritmos importantes.

Pseudocódigo Counting Sort

```
for i in 0 to (K - 1)
    counts[i] = 0

for each input number n
    counts[n] = counts[n] + 1

for i in 0 to (K - 1)
    assert( 0 <= counts[i] )
    for j in 0 to counts[i]
         output i
```

Pseudocódigo Quicksort

```
Quicksort(A as array, low as int, high as int){
    if (low < high){
        pivot_location = Partition(A,low,high)
        Quicksort(A,low, pivot_location)
        Quicksort(A, pivot_location + 1, high)
    }
}
Partition(A as array, low as int, high as int){
     pivot = A[low]
     leftwall = low

     for i = low + 1 to high{
         if (A[i] < pivot) then{
             swap(A[i], A[leftwall + 1])
             leftwall = leftwall + 1
         }
     }
     swap(pivot,A[leftwall])

    return (leftwall)}
```

Pseudocódigo Radix Sort

```
radix_sort(list):
    # Para cada posición de una llave:
    for pos from 0 to 10:
        # Para cada elemento de la lista:
        for every item in list:
            # Tomar el componente de la llave en la posición actual.
            element ⟵ get_element(item, pos)
            # Introducir el elemento en la jarra con su etiqueta.
            jars[element] ⟵ jars[element] + item;

        # Reunir todas las jarras en orden de acuerdo con su etiqueta.
        list ⟵ jars[0] + jars[1]  + jars[2]  + jars[3]
                       + jars[4]  + jars[5]  + jars[6]
                       + jars[7]  + jars[8]  + jars[9]
```

- Diseño de pruebas unitarias

GeneratorTest

Scene1(): *Una nueva cantidad generada de elementos (1000), de piso (0) y techo (10000)*

Método	Escenario	Entrada	Salida
testAlreadyOrdered()	*Scene1()*	Ninguna.	.El arreglo ya está ordenado y no se repiten números
testAlreadyOrderedRepeated()	*Scene1()*	Ninguna	El arreglo ya está ordenado

testReverseOrdered()	*Scene1()*	Ninguna	El arreglo está ordenado de mayor a menor sin repetición.
testReverseOrderedRepeated()	*Scene1()*	Ninguna	El arreglo está ordenado de mayor a menor
testRandomOrdered()	*Scene1()*	Ninguna	El arreglo no es null
testRandomRepeated()	*Scene1()*	Ninguna	El arreglo no es null
testPercentOrdered()	*Scene1()*	0.1	El arreglo no es null
testPercentOrderedRepeated()	*Scene1()*	0.1	El arreglo no es null

Scene2(): *Una nueva cantidad generada de elementos (10), de piso (0) y techo (100)*

Método	Escenario	Entrada	Salida
testAlreadyOrdered()	*Scene2()*	Ninguna.	.El arreglo ya está ordenado y no se repiten números

testAlreadyOrderedRepeated( )	*Scene2()*	Ninguna	El arreglo ya está ordenado
testReverseOrdered()	*Scene2()*	Ninguna	El arreglo está ordenado de mayor a menor sin repetición.
testReverseOrderedRepeated( )	*Scene2()*	Ninguna	El arreglo está ordenado de mayor a menor
testRandomOrdered()	*Scene2()*	Ninguna	El arreglo no es null
testRandomRepeated()	*Scene2()*	Ninguna	El arreglo no es null
testPercentOrdered()	*Scene2()*	0.5	El arreglo no es null
testPercentOrderedRepeated( )	*Scene2()*	0.5	El arreglo no es null

Scene3(): *Una nueva cantidad generada de elementos (100000), de piso (0) y techo (1000000)*

Método	Escenario	Entrada	Salida

testAlreadyOrdered()	*Scene3()*	Ninguna.	.El arreglo ya está ordenado y no se repiten números
testAlreadyOrderedRepeated()	*Scene3()*	Ninguna	El arreglo ya está ordenado
testReverseOrdered()	*Scene3()*	Ninguna	El arreglo está ordenado de mayor a menor sin repetición.
testReverseOrderedRepeated()	*Scene3()*	Ninguna	El arreglo está ordenado de mayor a menor
testRandomOrdered()	*Scene3()*	Ninguna	El arreglo no es null
testRandomRepeated()	*Scene3()*	Ninguna	El arreglo no es null
testPercentOrdered()	*Scene3()*	0.9	El arreglo no es null
testPercentOrderedRepeated()	*Scene3()*	0.9	El arreglo no es null

- Análisis de complejidad temporal y espacial.

Análisis de complejidad temporal Counting Sort

	Constante	Mejor caso	Peor caso
private void countingsort(int[] array) {		Temporal	
int[] aux = new int[array.length];	C1	1	
int min = array[0];	C2	1	
int max = array[0];	C3	1	
for (int i = 1; i < array.length; i++) {	C4	n	
if (array[i] < min) {	C5	n-1	
min = array[i];	C6	1	(n-1)/2
} else if (array[i] > max) {	C7	n-1	
max = array[i];	C8	1	(n-1)/2
}			
}			
int[] counts = new int[max - min + 1];	C9	1	
for (int i = 0; i < array.length; i++) {	C10	n+1	
counts[array[i] - min]++;	C11	n	
}			
counts[0]--;	C12	1	
for (int i = 1; i < counts.length; i++) {	C13	k	
counts[i] = counts[i] + counts[i-1];	C14	k-1	
}			
for (int i = array.length - 1; i >= 0; i--) {	C15	n+1	
aux[counts[array[i] - min]--] = array[i];	C16	n	
}			
this.array = aux;	C17	1	
}			
		Total	8n+5+k
	Complejidad	O(n)	

Análisis de complejidad espacial Counting Sort

Tipo	Nombre	Costo	Cantidad
Entrada	array	K1	n
Auxiliar	min	K2	1
	max	K2	1
	counts	K1	m
Salida	aux	K1	n
Complejidad O(n)			

Análisis de complejidad temporal Radix Sort

	Constante		
private int getMax(int arr[], int n)			
{			
int mx = arr[0];	C1	1	
for (int i = 1; i < n; i++)	C2	n	
if (arr[i] > mx)	C3	n-1	
mx = arr[i];	C4	n-1	
return mx;	C5	1	
}		total	3n

private void countSort(int arr[], int n, int exp)			
{			
int output[] = new int[n];	C6	1	
int i;	C7	1	
int count[] = new int[10];	C8	1	
Arrays.fill(count,0);	C9	1	
for (i = 0; i < n; i++)	C10	n+1	
count[(arr[i]/exp)%10]++;	C11	n+1	
for (i = 1; i < 10; i++)	C12	10	
count[i] += count[i - 1];	C13	9	
for (i = n - 1; i >= 0; i--)	C14	n-2	
{			
output[count[(arr[i]/exp)%10] - 1] = arr[i];	C14	n-3	
count[(arr[i]/exp)%10]--;	C15	n-3	
}			
for (i = 0; i < n; i++)	C16	n+1	
arr[i] = output[i];	C17	n	
}		total	7n+17

private void radixsort(int arr[], int n)			
{			
int m = getMax(arr, n);	C18	3n	
for (int exp = 1; m/exp > 0; exp *= 10)	C19	k-1	
countSort(arr, n, exp);	C20	7n+17	
}		total	10n+k+16
	Complejidad O(n)		

Análisis de complejidad espacial Radix Sort

Tipo	Nombre	Costo	Cantidad
Entrada	arr	K1	n
Auxiliar	mx	K2	1
	output	K1	n
	i	K2	1
	count	K1	10
	m	K2	1
Salida	aux	K1	n
Complejidad O(n)			