Estructuras de Datos.
Facultad de Ciencias, UNAM 2015-2.
Práctica 09: Ordenamientos.

Armando Ballinas Nangüelú.
armballinas@gmail.com

Fecha de entrega: 19 de mayo de 2015.
La práctica se puede hacer en parejas.

1. Introducción.

En esta práctica se implementarán varios algoritmos de ordenamiento y se comparará su rendimiento.

2. Algoritmos de ordenamiento.

Existen varios algoritmos de ordenamiento, en esta práctica se deben implementar cuatro escogiendo uno de cada subsección.

2.1. Con complejidad cuadrática.

Entre los algoritmos de complejidad cuadrática se encuentran:

- *Bubblesort*.
- *Insertion sort*.

Ambos fueron vistos en el curso y se debe implementar al menos uno de ellos.

2.2. *Quicksort*.

Quicksort fue inventado por sir. Anthony Hoare y aunque su complejidad en el peor caso es de orden cuadrático, en el caso esperado su complejidad es óptima. Este algoritmo es muy utilizado en la práctica y en esta será implementado. Este algoritmo también fue visto en el curso.

2.3. Con complejidad óptima.

La complejidad temporal del problema de ordenamiento es $\Theta(n \log n)$, donde n es el tamaño de la secuencia a ordenar. Hay varios algoritmos que tienen esta complejidad en el peor caso, entre ellos destacan:

- *Mergesort*.
- *Heapsort*.

Mergesort fue inventado por John Von Neumann y fue visto en clase. Por otro lado *Heapsort* es un algoritmo de complejidad óptima que se especifica a continuación.

2.4. Con complejidad lineal.

Aunque el problema de ordenación tiene la cota antes mencionada, si se restringe el tipo de elementos a ordenar de cierta manera se pueden crear algoritmos de complejidad lineal. En esta sección se mencionan dos de ellos y las restricciones que imponen a la cadena de entrada

- *Conting sort*. Este es un algoritmo de ordenamiento en donde los n números a ordenar son enteros entre 0 y k, con $k \in O(n)$, es decir se acota el posible valor de los números a ordenar en un rango cuyo tamaño es proporcional al tamaño de la entrada.
- *Bucket sort*. Este algoritmo de ordenamiento toma n números entre 0 y 1. Estos números se supone están uniformemente distribuidos entre $[0, 1)$.

  Se debe elegir uno de estos dos algoritmos para ser implementado.

3. Implementación.

Se debe elegir un algoritmo de cada tipo y *Quicksort* lo que quiere decir que se deben implementar al menos 4 algoritmos de ordenamiento. Para cada uno de ellos se debe crear una clase que tenga un método llamado `sort` que implemente el algoritmo en cuestión. El nombre de la clase debe ser el nombre del algoritmo que se implementa en ella. Los métodos `sort` deben tener la firma: `public T[] sort();` el arreglo que devuelven es el arreglo ordenado.

Además del método para ordenar, cada ejemplar de una clase debe ser construido con la secuencia a ordenar. Pueden suponer que esta secuencia es un arreglo de objetos de tipo `T` y cuya clase (T) implementa la interfaz `Comparable.java`. Con lo anterior, pueden usar el método `compareTo` de la interfaz.

4. Programas de prueba.

Se deben crear métodos `main` en donde se probarán los algoritmos. Cada algoritmo contará con su método main y la entrada será igual para todos. Como prueba para el programa solo se ordenarán números. Los números pueden ser flotantes o enteros. El programa recibirá como argumento una letra "e" indicando que trabajará con números enteros o una letra "f" indicando que lo hará con flotantes.

Además recibirán como argumento el nombre del archivo de texto que contiene los números a ordenar. Estos archivos contienen 1 millón de números y se incluyen con la práctica.

El programa deberá crear un ciclo. En cada iteración del ciclo se ordenarán los primeros $50000i$ números donde i es el número de la iteración y se procederá así hasta ordenar los primeros 500 mil elementos del archivo.

El programa deberá ordenar los números usando el algoritmo implementado en cada clase. Obsérvese que en el caso de la implementación de *Bucket sort* no se podrá trabajar con enteros y en el caso de *Counting sort* no se podrá trabajar con flotantes.

Para cada algoritmo de ordenamiento se debe medir el tiempo de ejecución y guardarse en un archivo aparte, es decir, tras la ejecución de cada método `main` se tendrá un archivo escrito. Cada archivo tendrá el nombre del algoritmo de ordenamiento y cada línea tendrá dos valores: el primero será el número de elementos ordenados y el segundo el tiempo en **milisegundos** que fue requerido para ordenar ese número de elementos. Por ejemplo el contenido del archivo "quicksort.txt" podrá verse así:

```
50000 0.002
100000 0.04
150000 0.09
200000 1.04
.
.
.
500000 124.15
```

Así mismo, solo para la última iteración, se debe crear un archivo en donde se guarden los números ordenados con cada ordenamiento.

En resumen:

1. Cada algoritmo tendrá un método `main` que recibirá dos argumentos: el primero es una letra "e" o una letra "f". El segundo es el nombre de un archivo de texto, en caso de recibir la "e" ese archivo deberá contener 1 millón de números enteros, en el caso de la "f" deberá contener un millón de números con punto flotante entre 0 y 1.

2. El programa ordenará los números incrementando el número de elementos de 50 mil en 50 mil tras cada iteración. En cada iteración se deben ordenar los números correspondientes asegurándose de hacer una copia de los mismos.

3. Para cada iteración y para cada algoritmo se escribirá el tiempo que tardó ese algoritmo en el archivo de texto correspondiente.

4. Para la última iteración se creará un archivo de texto por cada algoritmo donde se guardarán los números ordenados con ese algoritmo.

Los archivos que genere el programa deben ser:

- "bubblesort.txt" o "insertionsort.txt". Este archivo contendrá los resultados de la ejecución de este algoritmo. Un resultado por cada iteración anotando el número de elementos ordenados en esa iteración y el tiempo del algoritmo.

- 'quicksort.txt". Este archivo contendrá los resultados de la ejecución de este algoritmo. Un resultado por cada iteración anotando el número de elementos ordenados en esa iteración y el tiempo del algoritmo.

- "mergesort.txt" o "heapsort.txt". Este archivo contendrá los resultados de la ejecución de este algoritmo. Un resultado por cada iteración anotando el número de elementos ordenados en esa iteración y el tiempo del algoritmo.

- "countingsort.txt" o "bucketsort.txt". Este archivo contendrá los resultados de la ejecución de este algoritmo. Un resultado por cada iteración anotando el número de elementos ordenados en esa iteración y el tiempo del algoritmo.
- "bubblesort_ordered.txt" o "insertionsort_ordered.txt". Este archivo contendrá 500 mil puntos ordenados usando este algoritmo.
- "quicksort_ordered.txt". Este archivo contendrá 500 mil puntos ordenados usando este algoritmo.
- "mergesort_ordered.txt" o "heapsort_ordered.txt". Este archivo contendrá 500 mil puntos ordenados usando este algoritmo.
- "countingsort_ordered.txt" o "bucketsort_ordered.txt". Este archivo contendrá 500 mil puntos ordenados usando este algoritmo.

5. Gráficas de rendimiento.

Se deben entregar dos gráficas en donde se muestren los resultados obtenidos tras probar los algoritmos. En la primera se pondrán los resultados de los 3 algoritmos generales. La gráfica mostrará en una función cada algoritmo y sus tiempos. El eje de las "y" será el tiempo en milisegundos y el eje de las "x" el número de elementos.

La segunda gráfica será la comparación del algoritmo de orden lineal con restricciones en el tipo de datos a ordenar contra el algoritmo general de su preferencia. En esta gráfica solo habrá dos algoritmos y la especificación de los ejes es la misma que en la anterior.

6. Ejercicios.

1. (2pts.) Implementar una clase `Bubblesort.java` donde se implementé *Bubblesort* o una clase `Insertionsort.java` donde se implementé *Insertion sort*.
2. (2pts.) Implementar una clase `Quicksort.java` donde se implementé *quicksort*.
3. (2pts.) Implementar una clase `Mergesort.java` donde se implementé *Mergesort* o una clase `Heapsort.java` donde se implementé *Heapsort*.
4. (2pts.) Implementar una clase `Countingsort.java` donde se implementé *Counting sort* o una clase `Bucketsort.java` donde se implementé *Bucket sort*.
5. (2pts.) Implementar los métodos `main` de modo que se generen los tiempos de resultados de cada algoritmo y los archivos de texto mencionados arriba. También se deben generar las gráficas pertinentes.

7. Archivos a entregar.

- Cuatro clases de java donde se implementen cuatro algoritmos de ordenamiento con sus respectivos métodos `main`.

- Los cuatro archivos con los resultados obtenidos (los tiempos de ejecución).
- Las dos gráficas obtenidas a partir de sus resultados en formato `jpg` o `png`.

8. Puntos extra.

1. (1 pt extra) Realizar *quicksort* iterativo; es decir, sin recursión,
2. (1 pt extra) Realizar *mergesort* iterativo; es decir, sin recursión.