Desarrollo de algoritmo

Se solicita desarrollar 3 algoritmos de ordenamiento: Insertion sort, Merge sort, Quick sort y los métodos Arrays.sort () y Collections.sort().

Análisis: La tarea que se me asigno fue ocupar los métodos Arrays.sort() y Collections.sort(). El primer paso a seguir fue buscar información sobre estos métodos. Ya que en el curso de POO (Programación orientada a objetos) se ven estos métodos en la última parte del curso, sabía como usarlos pero no sabía en detalle el funcionamiento.

Arrays.sort():

- Proviene de la clase Arrays.
- Ordena el arreglo especificado en orden numérico ascendente.
- El algoritmo de ordenamiento es un Quicksort Dual-Pívot de Vladimir Yaroslavskiy, Jon Bentley y Joshua Bloch. Este algoritmo ofrece el rendimiento O (n log (n)) en muchos conjuntos de datos que hacen que otras unidades rápidas se degraden a rendimiento cuadrático, y suele ser más rápido que las implementaciones tradicionales (de un solo pivote) Quicksort.

Collections.sort():

- Proviene de la clase Collections.
- Este método compara los elementos dentro de una lista antes de ordenar implementando dos interfaces:
    - Comparable: Ordena la lista especificada en orden ascendente, según el orden natural de sus elementos. Todos los elementos en la lista deben implementar la interfaz comparable. Implementa el método compareTo().
    - Comparator: Ordena la lista especificada de acuerdo con el orden inducido por el comparador especificado. Todos los elementos en la lista deben ser mutuamente comparables usando el comparador especificado. Implementa el método compare().
- El algoritmo de ordenamiento es un mergesort estable, adaptable e iterativo que requiere muchas menos comparaciones que n log (n) cuando el conjunto de entrada está parcialmente ordenado, mientras que ofrece el rendimiento de un mergesort tradicional cuando el conjunto de entrada se ordena al azar. Si la lista de entrada está casi ordenada, la implementación requiere aproximadamente n comparaciones. Los requisitos de almacenamiento temporal varían desde una

pequeña constante para listas de entrada casi ordenadas hasta n / 2 referencias de objetos para listas de entrada ordenadas al azar.

Diseño: Luego de analizar los métodos pasamos a diseñar nuestra API. Para esto se toma el trabajo realizado en POO en la parte de genéricos del curso.

Para ordenar datos primitivos (enteros), creamos una clase llamada OrdenaArreglo la cual implementa un método que invoca al método arrays.sort.

Para ordenar datos no primitivos (basados en clases), creamos una clase llamada Alumno que contiene 4 atributos: apellido, nombre, edad y nota, la cual nos servirá para crear objetos de tipo Alumno y se procederá a ordenarlos por los distintos atributos que contiene la clase. Dentro de esta clase implementamos dos interfaces: comparable y comparator. La interface comparable implementa el método compareTo que compara en orden natural. En este caso el orden natural seria ordenar alumnos por su apellido. La interface comparator implementa el método compare que lo usaremos para comparar por nombre, por edad y por notas. Luego de esto creamos una clase llamada OrdenaCollections la cual implementa cuatro métodos para ordenar por apellido, por nombre, por edad y por nota. Estos 4 métodos invocan el método collections.sort para ordenar según corresponda.

Implementación: Se procedió a implementar lo descrito en el diseño en la primera iteración sin mayores inconvenientes. En los métodos que ordenan la clase alumno se crearon 4 clases aparte de la de alumno para llevar a cabo el ordenamiento. 3 clases que implementaban la interface comparator con el método compare para comparar por nombre, edad y nota, y otra que implementaba la los collections.sort según corresponda.

En una segunda iteración se cambió esa estructura y los métodos para comparar se fueron todos a la clase alumno (compara por apellido, por nombre, por edad y por nota). La clase OrdenaCollection quedo igual con sus métodos que invocan los métodos collections.sort según corresponda.

Gonzalo Villalobos Grebe.