Práctica 1 - Ordenación

Análisis y Diseño de Software, 2022

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

ETSI de Telecomunicación

Universidad Politécnica de Madrid

Introducción

En esta práctica, vamos a profundizar en el uso y la implementación de ordenación en Java. Crearemos un recomendador de películas. Utilizaremos los mismos datos y conceptos del laboratorio 1, y el código es similar (clase `Movie`, interfaz `Recommender`, etc.).

Consideraciones importantes:

- Se han separado los ficheros a modificar en paquetes individuales para cada ejercicio. Por ejemplo, los ficheros que se deben modificar en el ejercicio 1 están en `es.upm.dit.adsw.p1.ej1`.
- Esta práctica es autocontenida. Todos los ficheros y los datos necesarios están incluidos en el proyecto de la práctica.
- **No se deben eliminar ficheros, clases o métodos** de los que se incluyen en la práctica, salvo mención explícita. Hacerlo puede causar que la evaluación no funcione correctamente.
- **No se deben modificar las firmas de las funciones existentes**. Hacerlo puede causar que la evaluación no funcione correctamente.
- Las clases e interfaces pueden haber sufrido modificaciones menores con respecto al código del laboratorio 1.

La puntuación de la práctica será sobre 10 puntos, con la siguiente distribución:

- Ejercicio 1: 2 puntos
- Ejercicio 2: 2 puntos
- Ejercicio 3: 6 puntos

Objetivos

1. Implementar un algoritmo de ordenación
2. Implementar un recomendador optimizado
3. Desarrollar pruebas unitarias para evaluar las soluciones y su eficiencia

Diagrama de clases

Los elementos principales de esta práctica son los siguientes:

Diagrama de clases

Actividades

Ejercicio 1: Desarrollar un algoritmo de ordenación utilizando `compareTo`

En el ejercicio 2 del laboratorio 1, desarrollamos el método `compareTo` de la clase `Movie`. Como hemos visto en clase de teoría, este método es útil para implementar algoritmos de ordenación porque nos define el orden relativo de dos instancias (e.d., objetos) de una misma clase.

En este primer ejercicio, debemos completar la implementación de la clase `DefaultSorter`. Esta clase sólo tendrá un método, que admite un array de películas como parámetro y se encargará de ordenar el array de películas de acuerdo a su popularidad. Este método no debe devolver ningún valor. Para establecer el orden relativo de dos elementos, el método debe usar la implementación de `Movie.compareTo` que hicimos en el laboratorio 1. Por tanto, se debe copiar la implementación de ese método en la clase `Movie`.

La ordenación se puede realizar utilizando cualquiera de los algoritmos que se proponen en clase (Inserción, Selección, Quick Sort, Merge Sort).

Para asegurar que el funcionamiento del método implementado es el correcto, se recomienda realizar varias pruebas unitarias que comprueben que el algoritmo funciona adecuadamente con diferentes entradas. Para conseguir diferentes entradas, una opción es desordenar un array conocido (p.e., el resultante de cargar las películas en `metadata_top100.tsv`). Para cambiar el orden de un array se pueden utilizar los métodos de la biblioteca estándar de java tales como `Collections.shuffle`, o manualmente intercambiar elementos en el array mediante de forma análoga al método `swap` en las transparencias de ordenación. Una vez desordenado el array, se puede proceder a ordenarlo utilizando `DefaultSorter`, y a comprobar que el resultado está completamente ordenado.

Resumen de modificaciones:

- Método `sort` de `DefaultSorter` en `es.upm.dit.adsw.p1.ej1`

Ejercicio 2: Optimizar el recomendador `DefaultRecommender`

En el laboratorio 1, el último ejercicio consistió en desarrollar un algoritmo de recomendación de películas. El algoritmo más simple para esta tarea consistiría en realizar varias búsquedas sobre la lista completa de películas. Sin embargo, podemos conseguir una complejidad menor si nos aseguramos de que el conjunto de películas esté ordenado antes de realizar la búsqueda.

Para ello, vamos a completar la implementación de `DefaultRecommender`, de tal forma que su conjunto de películas esté ordenado, y que la recomendación

de películas en cualquier idioma (`recommend(int n)`) sea lo más rápida posible. Para la búsqueda para un solo idioma (`recommend(int n, String lang)`), utilizaremos el mismo conjunto ordenado, filtrando de la manera adecuada.

Nota: al realizar una recomendación **no se debe modificar la lista de películas disponibles**. Dos llamadas consecutivas a la función `recommend` deben dar el mismo resultado.

Nota: El recomendador devolverá el mayor número de películas posible, con un máximo de `n`. Es decir, puede que el tamaño de la lista recomendada sea menor que el pedido si no hay suficientes películas en la lista (en total, o para el idioma pedido).

Nota: Se deben evitar operaciones costosas innecesarias, tales como ordenar la lista de películas múltiples veces.

Resumen de modificaciones:

- En `DefaultRecommender` en `es.upm.dit.adsw.p1.ej2`:
  - Constructor
  - Método `recommend(int n)`
  - Método `recommend(int n, String lang)`

Ejercicio 3: Implementar un nuevo recomendador para idiomas

Al utilizar el recomendador, vemos que para algunos idiomas se devuelven pocas películas, o la calidad de las películas recomendadas es muy baja. Eso nos lleva a tener pocos resultados, o a que algunos de los resultados no sean muy buenos.

Tras pensarlo un poco, nos damos cuenta de que lo que realmente querríamos es un recomendador que siempre tenga en cuenta todas las películas, pero que pueda **priorice** las películas en el idioma elegido. Para ello, en lugar de utilizar la popularidad para ordenar, hemos ideado una nueva medida (que llamaremos **score** o puntuación de recomendación), que se calcula de la siguiente manera:

```
score = popularity * language_factor
```

Donde `language_factor` tendrá un valor de `1` para películas en el idioma elegido, y de `0.5` en el resto de casos. En otras palabras: cuando recomendemos para un idioma, trataremos las películas de ese idioma como si tuvieran el doble de popularidad que el resto de películas.

En principio, nos planteamos implementar el recomendador como en el Ejercicio 2. El problema es que ahora el orden de las películas cambiará en función del idioma de la recomendación. Eso nos obligaría a reordenar la lista en cada llamada al recomendador. En su lugar, decidimos enfocar el problema de forma diferente.

El recomendador va a separar las películas en listas separadas (que llamaremos cubos o `buckets`). Habrá una lista para cada uno de los idiomas en el conjunto

de películas. Así, habrá una lista con todas las películas en inglés, otro con todas las películas en español, etc. Cuando se nos pida una recomendación, tendremos seleccionar las películas a devolver, inspeccionando los cubos para cada idioma.

Implementaremos este recomendador en la clase `LanguageRecommender`, en el paquete `es.upm.dit.adsw.p1.ej3`. Vemos que ya se nos proporciona el método `recommend(int n)` al completo. Parece funcionar de una manera parecida al algoritmo MergeSort que hemos visto en clase. La primera tarea en este ejercicio será entender este código.

Por último, la implementación de `recommend(int n, String lang)` está vacía. Ayudándonos de la implementación del otro método, y de la definición de `score`, no debería resultarnos difícil completarla.

Resumen de modificaciones:

- método `recommend(int n, String lang)` de `LanguageRecommender` en `es.upm.dit.adsw.p1.ej3`:

Ejercicio 4 (Opcional): Implementar un nuevo recomendador mediante `Comparator`

Se aconseja complejar este ejercicio para entender cómo se consigue ordenar con diferentes criterios en Java. No obstante, este ejercicio es completamente opcional y no se verá reflejado en la nota.

En el ejercicio 3 hemos implementado el recomendador separando las películas por idioma y calculando el elemento a extraer en cada pasada. Como alternativa, podríamos ordenar todas las películas mediante su `score`, y devolver los primeros elementos de la lista, tal y como hicimos en el ejercicio 1.

El único problema es que en el ejercicio 1 la ordenación funciona porque implementamos el método `compareTo` para tener en cuenta sólo la popularidad. Si cambiamos la implementación para tener en cuenta el `score`, dejará de funcionar el código anterior que ordenaba por popularidad.

Java ofrece una alternativa para estos casos. A la hora de ordenar, en lugar de utilizar el método `compareTo`, podemos especificar al algoritmo un objeto que implemente el interfaz`Comparator,` que ofrece un método `compare(objeto1, objecto2)`. Su funcionamiento es análogo al de `compareTo`, salvo que no se invoca sobre una de las instancias. Esto nos permite tener varias clases que comparen elementos (películas en nuestro caso) de manera diferente, sin conflicto entre ellas.

Para completar este ejercicio, tendremos que terminar el desarrollo de dos clases:

- `LanguagePopularityComparator`: Un comparador de películas que usará el score (es decir, popularidad y el idioma)
- `CustomSorter`: una clase que ordena arrays de películas utilizando un comparador que se especificará en el constructor. La comparación se

hará usando el método `compare` de ese comparador, en lugar del método `compareTo` de la clase `Movie`. Nuevamente, se puede utilizar cualquier método de ordenación, pero se recomienda utilizar algoritmos de menor complejidad.

Adicionalmente, se proporciona la clase `LanguageRecommender`, que junta `CustomSorter` y `LanguagePopularityComparator` para dar una recomendadoción.

Para facilitar el desarrollo, la clase `es.upm.dit.adsw.p1.P1Tester` incluye una prueba de este recomendador, pero está comentada. Para probar nuestra implementación, sólo tenemos que descomentar este código.

Resumen de modificaciones:

- método `sort` de `CustomSorter` en `es.upm.dit.adsw.p1.ej4`:
- método `compare` de `LanguagePopularityComparator` en `es.upm.dit.adsw.p1.ej4`

Anexos:

Además de los anexos específicos de la práctica, se recomienda consultar de nuevo los anexos del laboratorio 1.

Anexo 1: Manejar Arrays, ArrayLists y Lists

Tanto en el laboratorio como en la práctica se usan diferentes tipos de estructuras de datos (contenedores) para representar un conjunto de películas. Algunos métodos aceptan como parámetro o devuelven un objeto del tipo `Movie[]` (array o vector), otros del tipo `List<Movie>` y otros objetos del tipo `ArrayList<Movie>`.

La mayor diferencia existe entre las listas (`List<Movie>`) y los arrays (`Movie[]`). Los arrays son conjuntos de un tamaño conocido (accesible mediante la propiedad `.length`), que generalmente se guardan en zonas contiguas en memoria. Para acceder al elemento en la posición `index` de un array llamado `pelis`, accederíamos de la siguiente manera: `pelis[index]`. Si la posición es incorrecta (menor que cero o mayor a la posición del último elemento), ese método lanzará una excepción. La sintaxis para modificar una posición del array es similar: `pelis[index] = nuevo_valor`, suponiendo que `nuevo_valor` sea una película válida.

Las listas de elementos son colecciones de tamaño variable, en las que se pueden añadir o eliminar elementos en cualquier punto de la lista. Hay muchas formas de implementar una lista, cada una con sus ventajas y sus inconvenientes, que las hacen aptas para diferentes usos. En el laboratorio 0 vimos una posible implementación: las listas doblemente enlazadas. Para permitir que existan diferentes implementaciones de manera transparente para los usuarios, la biblioteca estándar de Java define la interfaz `List<T>`, que es genérica sobre cualquier tipo de objeto que se quiera guardar. Esa interfaz expone los métodos básicos necesarios para, entre otros, añadir elementos (`add`), borrarlos (`remove`), reemplazarlos

(`set`) y comprobar el tamaño de la lista en un momento determinado (`size`). Además de la interfaz `List`, la biblioteca estándar de Java nos proporciona las implementaciones básicas más utilizadas. Entre ellas, tenemos `ArrayList`, que implementa internamente la lista mediante arrays, de un tamaño adecuado.

Como los arrays y las listas son estructuras muy habituales y fuertemente relacionadas, existen métodos para convertir listas en arrays, y viceversa. Dado un array, podemos convertirlo en una lista utilizando el método estático `Arrays.asList` de la clase Arrays en la biblioteca estándar de Java. Dada una lista, podemos utilizar el método `toArray` de la interfaz `List`. Este método acepta como argumento un array ya creado, o un método que se utilizará para inicializar el array en el que se colocarán los elementos. En nuestro caso, queremos que la inicialización sea crear un nuevo array de películas, por lo que usaríamos `Movie[]::new` como argumento.

Como ejemplo de uso, tenemos este código:

```
// Convierte una lista en un array
List<Movie> pelis = Movies.allFromFile("data/metadata_top100.tsv");
Movie[] lista = pelis.toArray(Movie[]::new);

// Borra el primer y último elementos
lista.remove(0);
lista.remove(lista.size()-1);

// Convierte un array de nuevo en una lista
List<Movie> nuevas_pelis = Arrays.asList(lista);
```

Nota: La conversión de Arrays a Listas (y viceversa) es una operación cuya complejidad tanto en tiempo como en espacio dependerá de la implementación específica de `List`. Por ejemplo, la conversión de/desde `ArrayList` es menos costosa que de/desde `LinkedList`. ¿Sabrías razonar por qué?. ¿Qué orden esperarías para cada una?.

Nota: en el método `toArray` utilizamos `Movie[]::new`, que es una referencia al método que crea un nuevo array de películas. Se podría pensar en usar `new Movie[]` como argumento, pero en ese caso lo que sucedería es que se intentaría crear un nuevo array (pero sin especificar tamaño, lo que es un error), que luego se intentaría pasar como argumento.

Anexo 2: Orden ascendente y descendente y límites en los Arrays/Listas

Hasta ahora, siempre hemos ordenado los elementos en orden ascendente. Por tanto, elementos menores aparecen primero en el array.

Por otro lado, nuestro recomendador tiene que devolver los elementos mayores que encuentre. Supongamos que tenemos que devolver `n` elementos, desde en una lista de tamaño `size`. Si ordenamos en orden ascendente, tendremos que

devolver desde el elemento `size-n` hasta el elemento `size`. Pero, ¿qué pasa si el `n` es mayor que `size`?, ¿haremos una sublista desde un índice negativo?. Eso no es posible y recibiríamos una excepción, así que tendremos que comprobar esas condiciones. La manera habitual es utilizar `Math.min` y `Math.max` para no permitir que los valores sobrepasen un cierto límite. Esto se conoce también como "saturar".

Así, en el caso que teníamos antes, podríamos tener el siguiente código:

```
// Si el valor de n es mayor que peliculas.size(), inferior tomará el valor 0.
int inferior = Math.max(0, peliculas.size()-n);
return peliculas.subList(inferior, peliculas.size());
```

Alternativamente, podríamos elegir ordenar descendentemente el array (o la lista). Así, en lugar de devolver los `n` últimos elementos, nos podemos centrar en los `n` primeros elementos. Para realizar este tipo de ordenación, hay varias opciones, por ejemplo:

- Ordenar de la forma habitual, y luego invertir el orden mediante `Collections.reverse(peliculas)`. Esto puede ser muy ineficiente, porque requiere mover todos los elementos.
- Definir el método `compare` o `compareTo` al revés, de forma que devuelva el valor contrario (p.e., 1 en lugar de -1 cuando `this.popularity` < `other.popularity`).
- No modificar los métodos `compare` o `compareTo` y definir un comparador personalizado (ver Ejercicio 4).
- No modificar los métodos `compare` o `compareTo` y usar `Collections.reverseOrder()`, que nos devuelve un comparador que funciona al revés que el comparador por defecto (`compareTo`).

Independientemente de cómo se haya ordenado el array (o la lista), tendremos que comprobar los límites de la sublista que pedimos, igual que hicimos cuando ordenamos ascendentemente. En este caso, el código sería parecido a este:

```
// Si el valor de n es mayor que peliculas.size(), inferior tomará el valor 0.
int superior = Math.min(peliculas.size(), n);
return peliculas.subList(0, superior);
```