

Práctica 1: algoritmos voraces: selección de actividades

Algoritmos Avanzados. Grado en Ingeniería Informática.



19 de octubre de 2017

Jorge Aranda García

Jose vicente bañuls garcía

# **ÍNDICE**

# INTRODUCCIÓN

El objetivo de esta práctica es permitirnos a los alumnos profundizar en el conocimiento de los algoritmos voraces y su funcionamiento. Para ello se ha propuesto el problema de la selección de actividades, en el que se busca una serie de actividades compatibles entre sí, es decir, cuyos tiempos de inicio y fin no se pisen, de tal forma que se escoja un número máximo. Para ello se han propuesto dos opciones. La primera: o bien el array de tiempos de comienzo se encuentra ordenado de mayor a menor, o bien el array de tiempos de fin se encuentra ordenado de menor a mayor. La segunda: ninguno de estos arrays se encuentra ordenado. Las soluciones propuestas para cada apartado son bien distintas, como se podrá ver en apartados posteriores.

# ENTORNO DE DESARROLLO

El entorno de desarrollo utilizado ha sido Eclipse Java Neon y la versión utilizada de JDK ha sido JDK 1.8.0\_60. El uso de este entorno de desarrollo se debe al alto grado de familiarización que tenemos con él, por lo que nos resulta sencillo de usar. Quizás la única desventaja con respecto a la herramienta propuesta, BlueJ, es la necesidad de crear una clase Main para poder ejecutar y comprobar cada método, aunque tampoco sea demasiado costoso.

# 

# ACTIVIDAD 1: VERSIÓN ORIGINAL

En esta actividad, la única precondición es que o el array de tiempos de comienzo se encuentra ordenado de forma decreciente o el array de tiempos de fin se encuentra ordenado de forma creciente. En nuestro caso, basándonos en el ordenamiento creciente de los tiempos de fin, la función de selección sería ir cogiendo cada vez el siguiente, empezando desde el primero hasta el último. El primero siempre va a ser solución, puesto que no se va a solapar con ninguna actividad anterior (aún no hay), por lo tanto, esa posición se marca a TRUE en el array de booleanos que servirá como solución. Se pasa a la siguiente actividad y si es compatible también se marca a TRUE. Si no a FALSE. Hay que tener en cuenta que este proceso se repite en todas las actividades, pero comparando con la última actividad escogida. Por ejemplo, si hemos cogido las actividades 0 y 1, la actividad 2 se compararía con la 1. En caso de que no sea compatible se pasaría a la actividad 3, que se volvería a comparar con la actividad 1. Hay que tener en cuenta al estar ordenadas las actividades, asumimos que el índice para cada actividad es su correspondiente posición en el array.

La cabecera del método que se corresponde con esta actividad es:

**public** **boolean**[] seleccionActividades(**int**[] c, **int** [] f){…}

En cuanto a la complejidad de este algoritmo depende directamente del bucle que recorre el array comprobando qué actividades son compatibles, con lo cual es de O(n), pero obliga al usuario a introducir el array de tiempos de fin ordenado.

# ACTIVIDAD 1.1: UN PASO INTERMEDIO

Esta actividad no ha sido explícitamente pedida. La justificación es que creímos oportuno un paso intermedio entre ambas actividades propuestas, además de ser una idea no demasiado compleja. Consiste en lo siguiente: En la segunda actividad se propone el uso del algoritmo en el caso de que ninguno de los arrays esté ordenado, pero se nos ocurrió una forma de ordenarlos para seguir usando la misma función de selección del apartado anterior de una forma tal, que el usuario no tenga que ordenar nada. Para ello recibíamos ambos arrays, y mediante un array de índices auxiliar, conseguíamos obtener la posición de cada tiempo de fin ordenado. El método de ordenación utilizado para el array de índices fue Shell. Este método compara cada elemento con el que está a un cierto número de posiciones llamado salto, en lugar de compararlo con el que está justo a su lado, como hacen otros métodos. Este salto es constante, y su valor inicial es N/2 (siendo N el número de elementos, y siendo división entera). Se van dando pasadas con el mismo salto hasta que en una pasada no se intercambie ningún elemento de sitio. Entonces el salto se reduce a la mitad, y se vuelven a dar pasadas hasta que no se intercambie ningún elemento, y así sucesivamente hasta que el salto vale 1. El porqué de usar este método es porque es uno de los pocos con complejidad menor de O(n2), además no es difícil de entender, ni “ensucia” demasiado el código. De esta forma obtenemos una solución parecida a la pedida en la actividad 2, de una forma parecida a la que se usa en la actividad 1. Además, de esta forma podemos tener un análisis más exhaustivo del algoritmo y poder así comparar tres variantes de éste: algoritmo con vector ordenado vs. algoritmo con vectores desordenador y ordenación interna vs. algoritmo con vectores desordenados.

El método que se corresponde con esta actividad es:

**public** **boolean**[] seleccionActividadesMejorado (**int** []c, **int** [] f){…}

La complejidad de este método depende de la complejidad del método de ordenación. Este método no ha sido escogido en vano, pues hemos intentado escoger un método que tenga la menor complejidad. Como ya se ha comentado , hemos escogido el método de ordenación SHELL , que tiene complejidad O(n2/3) promedio, que es ligeramente inferior al orden cuadrático de la mayoría de algoritmos de ordenación. A pesar de tener varios bucles con orden lineal, al no estar anidados la complejidad depende de la ordenación del array de índices.

# ACTIVIDAD 2: VERSIÓN FINAL

Tras haber afrontado las dos actividades anteriores, en el que de una forma u otra se obtenía algún tipo de ordenación sobre los arrays, surge la necesidad de implementar un algoritmo que encuentre una solución al problema que no necesite de los arrays ordenados. Para ello es necesario cambiar la función de selección. La función que nosotros hemos escogido consiste en lo siguiente: Escoger la actividad que tenga un tiempo menor final. Con respecto a esa primera, seleccionar la siguiente con el tiempo de fin menor y comprobar si es compatible. Cuando se encuentra una actividad que sea compatible y que además tenga el menor tiempo posible (para que la actividad sea compatible, antes han podido ser descartadas algunas actividades con tiempo de fin menor). De esta forma se obtiene una solución totalmente válida.

La cabecera del método que se corresponde esta actividad es:

**public** **int** seleccionActividadesSinOrden(**int**[] c, **int** [] f)

La complejidad de este método depende de dos bucles anidados uno que recorre los elementos para ir seleccionando candidatos y otro bucle que comprueba si el elemento seleccionado es de tiempo de fin mínimo. Por lo tanto la complejidad es de orden cuadrático.