# Gestión de sentencias: Proyecto de programación

Iván Matellanes Pastoriza
Asier Mujika Aramendia

*Con la colaboración de:*
Daniel Franco Barranco

*Asignatura:*
Estructuras de Datos y Algoritmos

*Grado en Ingeniería Informática*
*Diciembre de 2012*

# Índice

# 1. Resumen

Este documento trata de explicar en qué consiste el trabajo realizado y cuáles son los pasos que se han dado para alcanzar el resultado final.

Como se mostrará más adelante, el desarrollo de la aplicación se divide en tres partes, documentadas como *Versión 0, 1* y *2* respectivamente. Cada una de las partes corresponde a un punto del desarrollo en el que se realizó una evaluación del progreso con el profesor de la asignatura, Jesús Bermúdez. Se entiende que las versiones son aditivas, es decir, los métodos de la *Versión 0* se incluyen en la *Versión 1*, y los de la *Versión 1* en la *Versión 2*.

En cada apartado hemos querido plasmar las ideas, conceptos y decisiones tomadas a la hora de implementar los métodos, así como la explicación de los algoritmos utilizados y su análisis de complejidad computacional. También hemos considerado interesante medir los tiempos de ejecución de cada método y disponerlos en tablas para facilitar su lectura.

# 2. Introducción

## 2.1 Objetivos

La idea principal del proyecto que se nos presenta en esta asignatura es la de gestionar sentencias simples, que tendrá la forma de ***sujeto propiedad objeto***. Para realizar este cometido, las sentencias han de agruparse en almacenes sobre los que se definen una serie de operaciones a implementar. A continuación se listan algunos ejemplos de estas operaciones:

- Obtener las sentencias de un almacén con un sujeto determinado.
- Obtener una colección ordenada de todas las sentencias del almacén.
- Dar una colección de entidades que son sujetos en varios almacenes distintos.

- Cargar sentencias a un almacén desde un fichero de texto.

Algunas de las propiedades pueden recibir un tratamiento especial, como es el caso de ***es*** y ***subClaseDe***, y en base a este se piden ciertas operaciones, entre las que se incluyen: las clases que son superclases de otra, los profesores que trabajan para un determinado departamento, etc.

Desde el primer momento, la intención de los tres componentes del grupo ha sido realizar la estructuración e implementación del proyecto de la forma más eficiente posible en base a las estructuras de datos que conocemos. De este modo, hemos pretendido escribir código limpio, correcto y eficaz para la versión inicial o versión 0, y así evitar futuras restructuraciones o amplias modificaciones en el código. Las ideas planteadas se exponen más detalladamente en el siguiente apartado.

Éramos conscientes de que según avanzara el curso sería imprescindible realizar ajustes para hacer el programa más competitivo, pero nuestra intención era que los inevitables ajustes no supusieran replantear toda la estructura inicial.

## 2.2 Metodología

Realizar un trabajo en equipo puede ser una ardua tarea si no se administra debidamente. Es por ello que se decidió ir repartiendo la tarea según se avanzaba en el desarrollo del proyecto, procurando no solapar el trabajo de unos y otros y equilibrar la carga de trabajo de cada miembro del grupo.

Con objeto de facilitar el trabajo en equipo y agilizar la tarea de compartir el código del proyecto, se decidió abrir un repositorio común on-line del sistema de control de versiones *Git*. Esto nos ha facilitado la tarea de comunicar los cambios que se van realizando en el proyecto mediante *commits* y nos ha permitido a todos trabajar sobre el mismo código. Cabe decir que el repositorio es de acceso público y se encuentra en la siguiente dirección:

https://github.com/Sulley38/ProyectoEDA

# 3. Versión 0

## 3.1 Estructura

La idea básica con la que comenzamos a abordar este proyecto se detalla a continuación.

En primer lugar, para contener las relaciones entre sentencias, en lugar de utilizar *arrays* simples o listas enlazadas (que hubiera sido la idea más sencilla a primera vista), decidimos representar todas las sentencias utilizando un grafo dirigido ponderado en el que los nodos serían los sujetos y objetos, y el peso de las aristas estaría asociado a las propiedades de las sentencias. Se le asigna a cada entidad (sujeto, objeto y propiedad) un valor numérico para posteriormente administrar todo con una lista de adyacencia.


***Imagen 1.*** *Representación del grafo.*

Para relacionar el *string* de una entidad con su valor numérico correspondiente utilizamos un árbol de prefijos (trie).

Como en el ejemplo se puede observar, para obtener el valor de una palabra recorreremos el árbol siguiendo el camino del *string* letra a letra. En nuestro caso, haremos el mismo árbol, guardando sus características pero teniendo en cuenta que si el nodo no es el final de una palabra, tendrá el valor por defecto de -1. El valor que se indica en azul es el correspondiente al *string* que termina en ese nodo.

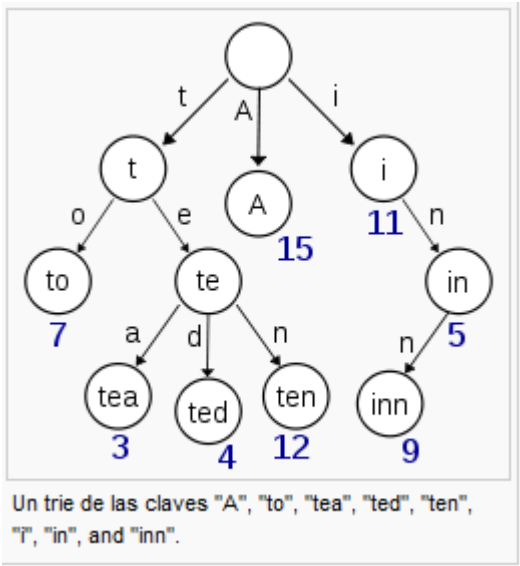


***Imagen 2.*** *Ejemplo de trie (Wikipedia).*

En el sentido contrario, si queremos conocer a qué *string* pertenece un índice dado, se accede a un *array* de *strings* en la posición del índice, que devuelve el *string* correspondiente.

Este sería un ejemplo completo de la estructura de "codificación/descodificación":

| 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|
| aa | bb | ba | a | ab | .... |

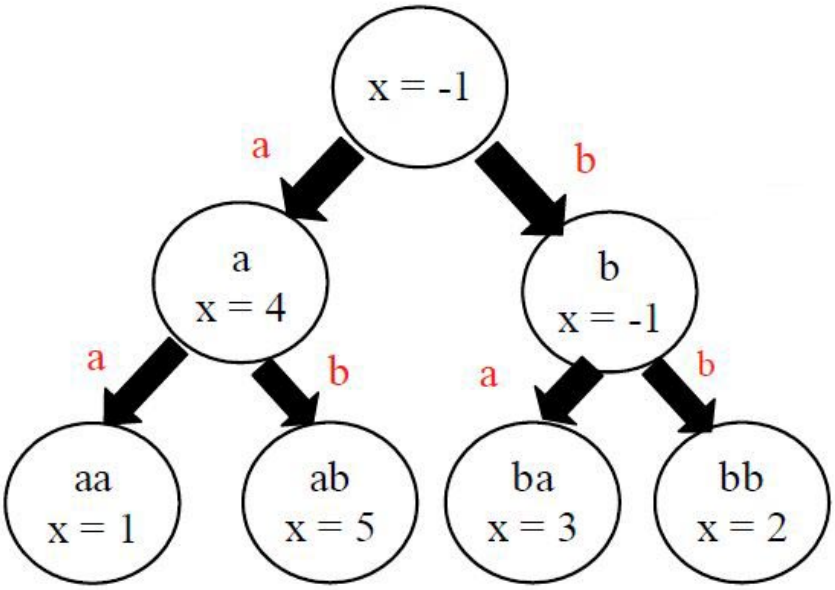


***Imagen 3.*** *Conversión int->string y string->int.*

## 3.2 Implementación

Con todos los datos ya en forma numérica, es turno de ver la implementación del grafo a nivel de código. En primer lugar, se pensó describirlo utilizando una matriz de adyacencia como la que se muestra en el siguiente ejemplo:

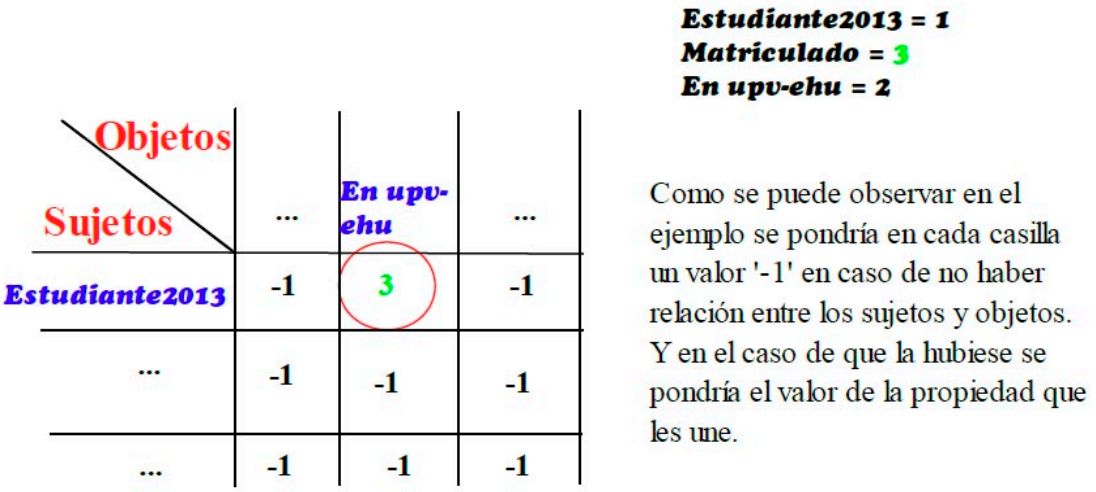


***Imagen 4.*** *Representación con matriz de adyacencia.*

Esta forma de organizar los objetos, sujetos y propiedades no resultó ser como creíamos de efectiva al tener que hacer una matriz de adyacencia con gran cantidad de filas y columnas y mucha memoria desaprovechada con valores -1. Por lo tanto, para esta tarea en la 3ª reunión decidimos utilizar otra estructura.

La nueva idea se basó en el concepto de lista de adyacencia. Hemos usado una lista (implementada como array) para las aristas de entrada a un nodo y otra para las salidas. Cada posición del array se corresponde con uno de los sujetos u objetos, siguiendo el valor que le otorgamos al introducirlo en el trie. Los valores se dan de forma secuencial, según el orden en que se leen desde el fichero de sentencias, y se han utilizado tries distintos para sujetos/objetos y propiedades. Cada posición del array tiene un apuntador al comienzo de una lista enlazada, cuyos elementos contienen 3 variables para darnos la siguiente información:

- Propiedad: nos da la propiedad que entra/sale del nodo en el que estemos.

- Objetivo: nos da la información del nodo al que va/del cual proviene.
- Repeticiones: nos da la información de cuantas veces está repetida la arista que entra/sale a este nodo.

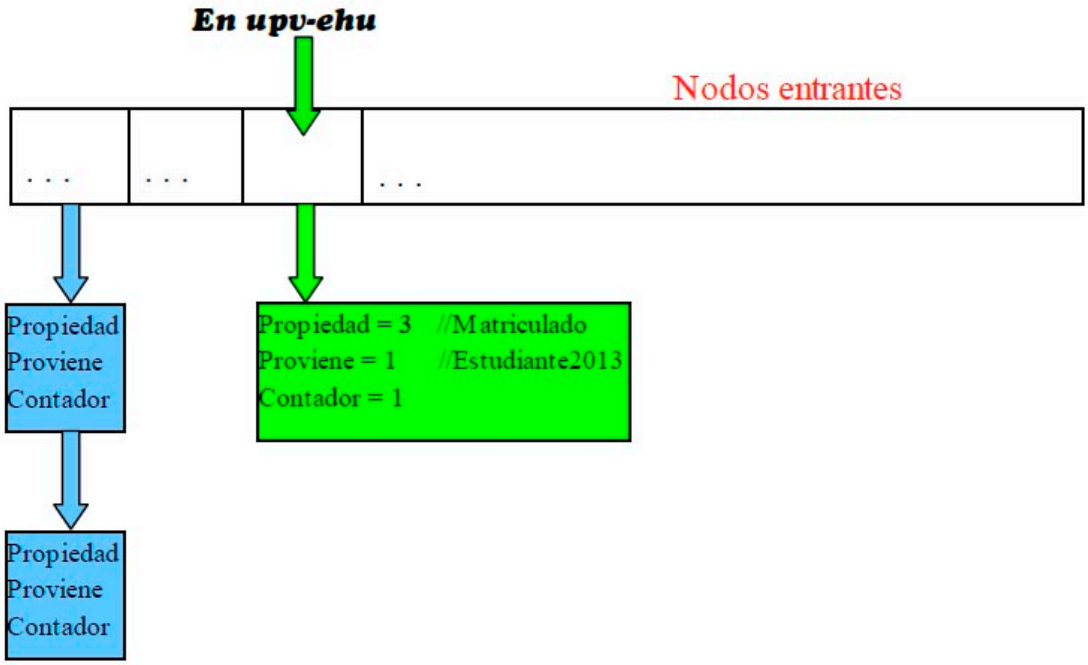


***Imagen 5.*** *Representación con lista de adyacencia.*

La misma idea que se muestra en la imagen 5 se aplica para la segunda lista de adyacencia, pero esta vez indicando las aristas que salen de cada nodo.

## 3.3 Análisis de complejidad

Las estructuras de "codificación/descodificación" permiten realizar ambas operaciones asociativas en un tiempo mínimo, como se puede apreciar en el análisis de complejidad de los métodos del árbol de prefijos:

Insertar un string en el trie:

```
public int insertar( String s, int valor ) { ... }
```

Avanza por las ramas del trie siguiendo el camino que dictan los caracteres de s. Si faltan nodos, se añaden y al último se le da el valor del parámetro. Si ya existen, se devuelve el valor del último nodo.

- *Tamaño de la entrada:* longitud de s ≡ l
- *Operación elemental:* asignación
- *Número de operaciones elementales en el caso peor:*

$$2 + \sum_{i=0}^{l-1} 2 + 2 = 2 + 2(l - 1 + 1) + 2 = \ 2l + 4$$

La longitud del string se puede considerar constante en el contexto de la práctica, luego:

$$O(1)$$

Obtener el valor numérico de un string del trie:

```
public int obtenerValor( String s ) { ... }
```

Avanza por las ramas del trie siguiendo el camino que dictan los caracteres de s. Devuelve el valor del último nodo visitado. Si falta algún nodo, la palabra no está en el trie y devuelve -1.

- *Tamaño de la entrada:* longitud de s ≡ l
- *Operación elemental:* asignación
- *Número de operaciones elementales en el caso peor:*

$$1 + \sum_{i=0}^{l-1} 1 = 1 + l - 1 + 1 = \ l + 1$$

La longitud del string se puede considerar constante en el contexto de la práctica, luego:

$$O(1)$$

Usando estos métodos, las cuatro primeras operaciones que se piden en el enunciado de la práctica resultan de un orden de complejidad bastante asequible:

1) Colección de sentencias del almacén que tienen un sujeto determinado:

```
public ListaEnlazada<String> sentenciasPorSujeto( String sujeto ) { ... }
```

Obtiene el valor *int* a partir del parámetro sujeto, y recorre la lista enlazada que se encuentra en la posición obtenida del array *nodosSalientes*, devolviendo todas las sentencias que indican los elementos de la lista.

- *Tamaño de la entrada:* longitud de sujeto ≡ l, número de aristas salientes del nodo ≡ a, número de veces que aparece repetida una arista determinada ≡ r
- *Operación elemental:* asignación
- *Número de operaciones elementales en el caso peor:*

$$2+\sum_{i=0}^{l-1}1+2+\sum_{it=1}^{a}\left(1+\sum_{i=0}^{r-1}3\right)=2+(l-1+1)+2+\sum_{it=1}^{a}(1+3(r-1+1))$$
$$=4+l+\sum_{it=1}^{a}3r+\sum_{it=1}^{a}1=4+l+3r(a-1+1)+(a-1+1)$$
$$=4+l+(3r+1)a$$

La longitud del sujeto se puede considerar constante en el contexto de la práctica, luego:

$$O(ar)$$

2) Colección de sentencias distintas del almacén que tienen un sujeto determinado:

```
public ListaEnlazada<String> sentenciasDistintasPorSujeto( String sujeto )
                                                    { ... }
```

Realiza las mismas operaciones que la función anterior, pero devolviendo cada sentencia una única vez.

- *Tamaño de la entrada:* longitud de sujeto ≡ l, número de aristas salientes del nodo ≡ a
- *Operación elemental:* asignación
- *Número de operaciones elementales en el caso peor:*

$$2+\sum_{i=0}^{l-1}1+2+\sum_{it=1}^{a}(1+3)=2+(l-1+1)+2+4(a-1+1)=4+l+4a$$

La longitud del sujeto se puede considerar constante en el contexto de la práctica, luego:

$$O(a)$$

3) Colección de propiedades distintas que aparecen en las sentencias del almacén:

```
public ListaArray<String> propiedadesDistintas() { ... }
```

Devuelve el array de propiedades.

- *Tamaño de la entrada:* -
- *Operación elemental:* retorno
- *Número de operaciones elementales en el caso peor:* 1

$$O(1)$$

4) Colección de entidades distintas que son sujeto de alguna sentencia y también son objeto de alguna sentencia de ese almacén:

```
public ListaEnlazada<String> entidadesSujetoObjeto() { ... }
```

Recorre las dos listas de adyacencia, y en caso de que en una misma posición ninguna esté vacía (un nodo tenga aristas entrantes y salientes), devuelve la entidad correspondiente a esa posición.

- *Tamaño de la entrada:* número de sujetos y objetos ≡ n
- *Operación elemental:* asignación
- *Número de operaciones elementales en el caso peor:*

$$1 + \sum_{i=0}^{n-1} 3 = 1 + 3(n - 1 + 1) = 1 + 3n$$

$$O(n)$$

El método más costoso y esencial de nuestra implementación es, sin duda, la constructora de la clase *Almacén*. Dada su importancia, se detalla el análisis del algoritmo a continuación:

```
public Almacen( String nombreDeArchivo ) { ... }
```

Inicializa los atributos de clase, lee las sentencias del fichero especificado y las inserta en las diferentes estructuras de datos del modo en que se ha explicado en el apartado anterior.

- *Tamaño de la entrada:* número de sentencias en el fichero ≡ s, longitud del sujeto ≡ $l_1$, longitud de la propiedad ≡ $l_2$, longitud el objeto ≡ $l_3$
- *Operación elemental:* asignación
- *Número de operaciones elementales en el caso peor:*

$$12 + \sum_{i=1}^{s}\left(4 + l_1 + 11 + l_2 + 5 + l_3 + 11 + 2\left(3 + \sum_{it=1}^{i} 3 + 3\right)\right)$$

$$= 12 + \sum_{i=1}^{s}\left(31 + l_1 + l_2 + l_3 + 2\big(6 + 3(i - 1 + 1)\big)\right)$$

$$= 12 + \sum_{i=1}^{s}(43 + l_1 + l_2 + l_3 + 6i)$$

$$= 12 + (43 + l_1 + l_2 + l_3)s + 6\left(\frac{1+s}{2}s\right)$$

$$= 12 + (46 + l_1 + l_2 + l_3)s + 3s^2$$

$$O(s^2)$$

La complejidad de las operaciones de estructuras tipo lista enlazada y *arrays* se ha comentado en clase, por lo que no consideramos necesaria su inserción en este documento.

## 3.4 Tiempos de ejecución

Para simplificar las pruebas y hacer más claro qué archivos contienen más datos, hemos decidido renombrar los archivos como A0, A1,..., A6 siendo A6.txt el más grande (*datosUniversidad99*) y A0.txt el archivo de pruebas más pequeño (*Extracto de datos para hacer pruebas*).

Todos los tiempos se tomaron en una máquina con procesador Intel i7 a 2.67 Ghz sobre arquitectura x86_64, 6 GB de memoria RAM, bajo el S.O. Windows 7. Se ejecutaron todas las instrucciones diez veces y en la tabla se ha plasmado el valor medio de estas mediciones.

| | **Sentencias de sujeto determinado** | **Sentencias distintas de sujeto determinado** | **Propiedades distintas del almacén** | **Entidades que son sujeto y objeto** | **Constructora del almacén** |
|---|---|---|---|---|---|
| **A0.txt** | 0.071299 ms | 0.042181 ms | 0.000919 ms | 0.018083 ms | 21.355142 ms |
| **A1.txt** | 0.048005 ms | 0.043867 ms | 0.000804 ms | 0.087735 ms | 34.169811 ms |
| **A2.txt** | 0.024366 ms | 0.021148 ms | 0.000459 ms | 0.67529 ms | 75.093962 ms |
| **A3.txt** | 0.025592 ms | 0.022106 ms | 0.000536 ms | 1.285565 ms | 1381.1628 ms |
| **A4.txt** | 0.034634 ms | 0.022604 ms | 0.000498 ms | 1.881511 ms | 4022.8261 ms |
| **A5.txt** | 0.021569 ms | 0.018543 ms | 0.000536 ms | 2.811501 ms | 12055.41 ms |
| **A6.txt** | 0.006053 ms | 0.003371 ms | 0.000344 ms | 6.106348 ms | 52886.36 ms |

Nota: el sujeto utilizado como argumento de los dos primeros métodos fue *<http://swat.cse.lehigh.edu/onto/univ-bench.owl#AdministrativeStaff>*

# 4. Versión 1

## 4.1 Cambios en la estructura

En primer lugar, la modificación más evidente y que no se llevó a cabo en la versión 0 por falta de tiempo ha sido en la implementación del trie. El atributo de cada nodo del árbol que contiene los punteros a los nodos hijos pasa de ser un array estático de 128 posiciones a una lista enlazada. Las ventajas de este cambio saltan a la vista: se aprovecha mucho mejor el espacio (dado el formato de las sentencias, muchas veces los nodos sólo tienen un hijo) y se puede usar cualquier carácter Unicode. La búsqueda de un nodo hijo se vuelve más lenta, pero en muchos casos es una pérdida de rendimiento despreciable debido al motivo antes expuesto: muchos nodos con un único hijo.

El segundo cambio importante se ha dado en la estructura de lista enlazada. Con vistas a la implementación del método de devolver sentencias en orden lexicográfico, se añade a la lista enlazada un método para ordenar sus elementos según un orden establecido mediante el método *compareTo* de la interfaz *Comparable<T>*. Esta ordenación se aplica en las listas de adyacencia.

Sin embargo, la pérdida de rendimiento con esta nueva estructura es notable, pues la ordenación en listas enlazadas es una operación de alta complejidad, pudiendo llegar a ser de orden cúbico.

Tras estudiar las posibilidades, se ha optado por convertir los elementos de las listas de adyacencia en listas de *arrays* estáticos, que permiten insertar elementos en orden en tiempo logarítmico gracias a la búsqueda dicotómica. Esto conlleva rescribir la estructura *ListaArray* para hacerla más flexible: como se desconoce el tamaño máximo que puede llegar a tener la lista, en caso de querer insertar un elemento en una lista llena, se amplía su tamaño máximo creando un nuevo array más grande y copiando los elementos del array viejo en el nuevo. El impacto en rendimiento de esta funcionalidad pasa desapercibido pues el nuevo algoritmo de ordenación mejora con creces el anterior.

Pese a la ganancia de rendimiento, éramos conscientes de que el

algoritmo seguía siendo mejorable. Por este motivo, decidimos darle otra vuelta más al problema. Hemos llegado a la conclusión de que es conveniente hacer la ordenación de las aristas en una lista aparte de la lista de adyacencia del grafo. Gracias a este nuevo mecanismo, mantenemos por un lado el orden de las aristas según se leen del fichero (igual que en la versión 0) y por otro lado, se tiene una réplica de la lista de adyacencia, pero con las aristas de cada nodo ordenadas alfabéticamente. La ordenación se hace en la constructora de almacén mediante el algoritmo *insertion sort*, suficientemente eficiente para listas de tamaño pequeño como es el caso (menos de 25 elementos).

Para no sacrificar mucha memoria al mantener dos listas de adyacencia, la segunda es una lista de índices que apuntan a las aristas del primero, como se muestra en este esquema:

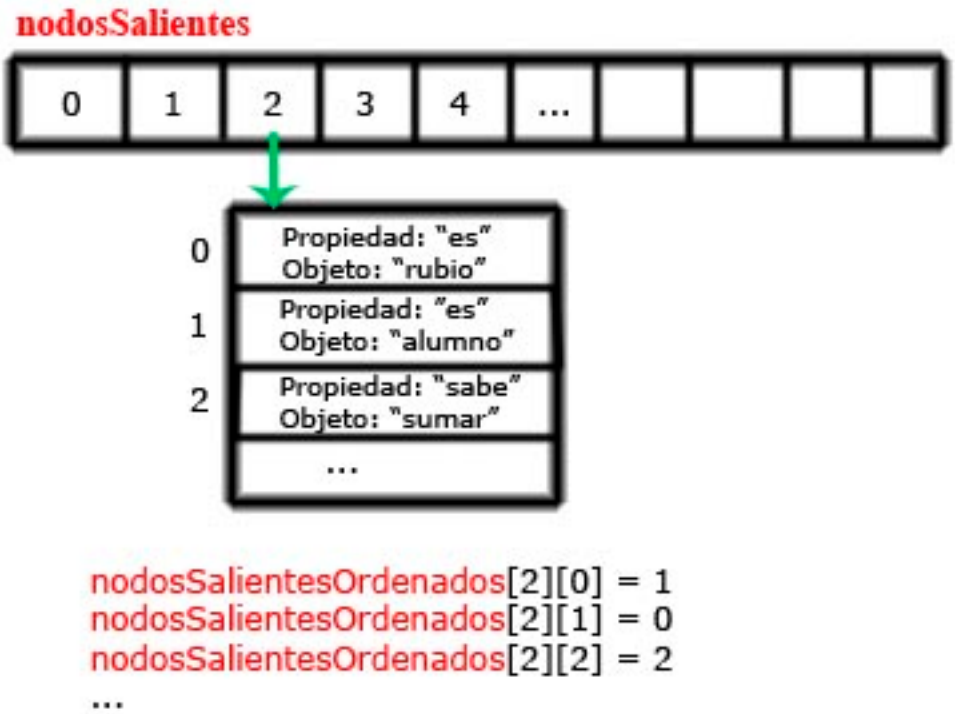


***Imagen 6.*** *Esquema actual de lista de adyacencia con arrays (nodosSalientes) y lista auxiliar de índices ordenada (nodosSalientesOrdenados).*

Tal y como se ve en la imagen, *nodosSalientesOrdenados[i][j] = k* significa que la *j*-ésima arista saliente ordenada del nodo *i*-ésimo sería la arista que está en la posición *k* en la lista de adyacencia original.

## 4.2 Nuevos métodos

El método 6 del enunciado es el primero en ser implementado. Con las estructuras de lista adaptadas para ordenar elementos con *insertion sort*, las propiedades y objetos se guardan ordenadas en listas auxiliares, de modo que lo único que queda es recorrer la estructura de trie en profundidad para tener los sujetos ordenados alfabéticamente.

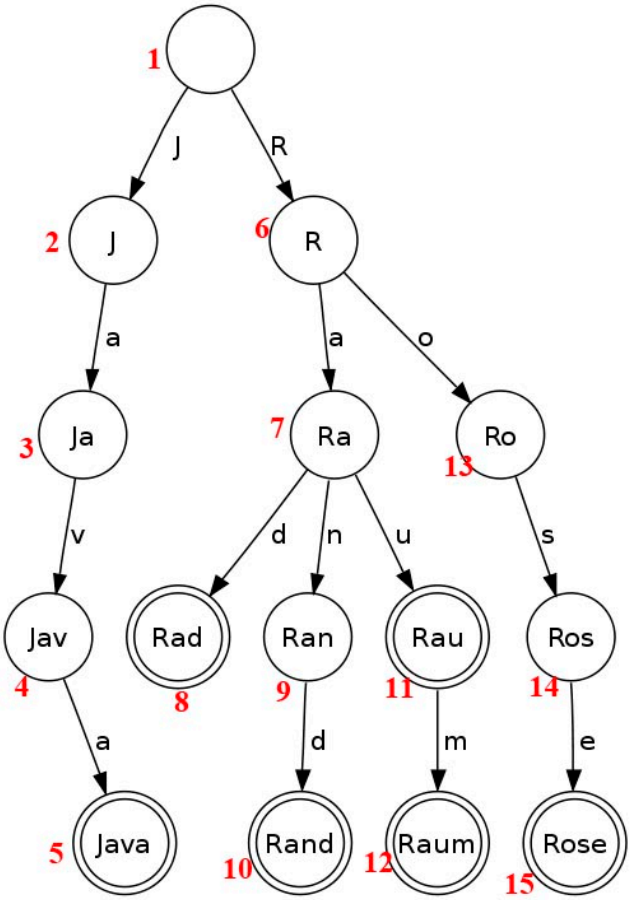


***Imagen 7.*** *En rojo, el orden en el que se visitarían los nodos.*

Este cometido se consigue mediante la función recursiva DFS:

Búsqueda en profundidad en el trie:

```
private void DFS( final NodoTrie n, final ListaArray<Integer> lista ) { ... }
```

Recorre el trie desde el nodo *n* e inserta los valores de los nodos visitados en *lista*.
Caso base (el nodo no tiene hijos): inserta su valor en *lista*.
Caso general (el nodo tiene al menos un hijo): inserta su valor en *lista* y llama a esta misma función para cada nodo hijo.

- *Tamaño de la entrada:* número de nodos del trie ≡ t
- *Operación elemental:* asignación
- *Número de operaciones elementales:* tres asignaciones por nodo: *3t* asignaciones

# O(t)

La ordenación de la lista de adyacencia se realiza en la constructora de la clase Almacén, por lo que su complejidad aumenta. En concreto, se hace un *insertion sort* por cada nodo. El método *sort* requiere $a^2$ comparaciones para ordenar $a$ aristas en el peor caso (como se ha visto en clase), y siendo $n$ el número de nodos del grafo, son necesarias $na^2$ operaciones elementales extra en la constructora, que se suman a las anteriores y dan una complejidad computacional de **$O(s^2 + na^2)$**, donde s representaba el número de sentencias en el fichero a leer.

Finalmente, se define una función tal y como se especifica en el enunciado:

6) Colección ordenada de todas las sentencias que aparecen en el almacén:

```
public ListaArray<String> sentenciasOrdenadas() { ... }
```

Devuelve una lista con todas las sentencias del almacén en orden lexicográfico.
Recorre la trie de sujetos en profundidad, y por cada nodo recorrido, devuelve las aristas que salen de él en la lista de adyacencia ordenada alfabéticamente.

- *Tamaño de la entrada:* número de nodos del trie ≡ t, número de aristas salientes del nodo ≡ a, número de veces que aparece repetida una arista determinada ≡ r
- *Operación elemental:* asignación
- *Número de operaciones elementales en el caso peor:*

$$3t + \sum_{i=0}^{t-1}\left(\sum_{j=0}^{a-1}\left(1 + \sum_{k=0}^{r-1} 3\right)\right) = 3t + \sum_{i=0}^{t-1}\left(\sum_{j=0}^{a-1}(1+3r)\right) = 3t + \sum_{i=0}^{t-1}(a+3ar)$$
$$= 3t + ta + 3tar$$

El número de nodos válidos en el trie y las repeticiones de una arista se pueden entender constantes puesto que, en general, guardan una gran diferencia respecto al número de aristas salientes de un nodo. La complejidad resultante sería:

# O(a)

Por otro lado, el método 5 se implementa de la siguiente manera:

5) Dada una colección de almacenes: Colección de entidades distintas que son sujetos en todos y cada uno de esos almacenes:

```
public static ListaEnlazada<String> entidadesSujetoEnTodos(Almacen[]
                        coleccionAlmacenes) { ... }
```

Devuelve una lista con todos los sujetos que están en todos y cada uno de los almacenes.
Encuentra el almacén con menor cantidad de sujetos (**O(v)**). Luego, recorre la lista *nodosSalientes* de este (**O(n)**) y por cada elemento comprueba que sea un sujeto, es decir, que *nodosSalientes* en esa posición no sea vacía (**O(1)**). En caso de ser sujeto, se obtiene el string correspondiente al nodo en cuestión, y con este string se comprueba si el sujeto está en el resto de almacenes usando sus respectivas tries (**O(l)** por cada almacén distinto al de menor tamaño). En caso de estar en todos, se inserta en la lista resultado.

- *Tamaño de la entrada:* número de almacenes pasados como parámetro ≡ v, número de sujetos y objetos en el almacén más pequeño ≡ n, longitud del sujeto a comprobar ≡ l
- *Operación elemental:* asignación/comparación
- *Número de operaciones elementales en el caso peor:*

$$2 + v + \sum_{i=0}^{n-1}\left(1 + 1 + 1 + \sum_{j=0}^{v-1}\left(\sum_{k=0}^{l-1} 1\right) + 1\right)$$

$$= 2 + v + \sum_{i=0}^{n-1}(4 + \mathrm{lv}) = \; 2 + v + (4 + \mathrm{lv})n = 2 + v + 4n + lvn$$

La longitud del sujeto a comprobar se puede considerar constante en el contexto de la práctica, y el número de almacenes pasados por parámetro no es significativo respecto a *n*, luego:

$$\mathrm{O(n)}$$

## 4.3 Tiempos de ejecución

| | **Sentencias en orden lexicográfico** | **Sentencias que están en el almacén cargado, en el A1 y A2** | **Constructora del almacén** |
|---|---|---|---|
| **A0.txt** | 1.296441 ms | 0.461315 ms | 1 ms |
| **A1.txt** | 2.164898 ms | 6.715946 ms | 3 ms |
| **A2.txt** | 14.905334 ms | 0.232975 ms | 59 ms |
| **A3.txt** | 62.634427 ms | 0.239373 ms | 724 ms |
| **A4.txt** | 118.029949 ms | 0.143708 ms | 2098 ms |
| **A5.txt** | 323.333483 ms | 0.143555 ms | 6550 ms |
| **A6.txt** | 1193.792837 ms | 0.017853 ms | 27559 ms |

El contexto en el que se han realizado las mediciones es el mismo que el citado en el apartado 3.4. Los tiempos de la constructora del almacén aparecen redondeados a la unidad.

# 5. Versión 2

## 5.1 Cambios en la estructura

Objetivo cumplido. El propósito inicial del grupo consistía en evitar cambios en la estructura básica de nuestro almacén de sentencias. Gracias a la representación de grafo que se escogió inicialmente, ha sido posible implementar los once métodos del enunciado de manera eficiente y sin tener que renunciar a esta estructura ni añadir nuevas (salvando los pequeños ajustes que se realizaron al comenzar a desarrollar la versión 1).

## 5.2 Nuevos métodos

Los cuatro métodos que faltan por implementar poseen una característica común: en todos ellos se pide, dado un nodo, buscar una serie de nodos conectados a través de aristas de peso determinado, que pueden ser las propiedades *es*, *subClaseDe*, *cursa*, etc.

El algoritmo general para resolver estos problemas va a ser la búsqueda en profundidad o en anchura. En nuestro caso, hemos elegido la búsqueda en profundidad para no tener que usar una estructura adicional de cola. En diversos casos es necesario utilizar la segunda lista de adyacencia (*nodosEntrantes*) que hasta ahora no había hecho falta para nada.

El siguiente dibujo muestra un ejemplo del recorrido a llevar a cabo en el método 10 dado el nodo 4 como parámetro, y cuyo resultado sería [2,7]:

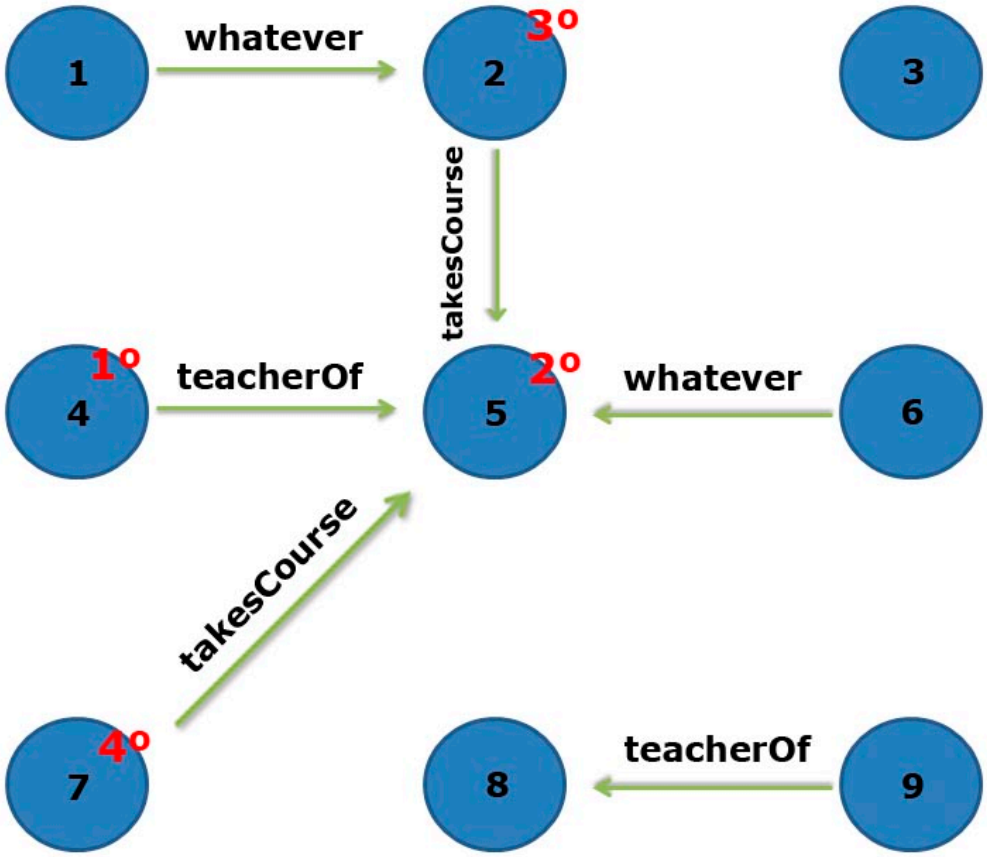


***Imagen 8.*** *En rojo, el orden en el que se visitarían los nodos.*

## 5.3 Análisis de complejidad

Como ya se ha explicado, los métodos son casos especiales de búsqueda en profundidad:

7a) Colección de las clases de un sujeto:

```
public ListaEnlazada<String> clasesDe(String sujeto) { ... }
```

Obtiene el valor numérico de sujeto, inicializa un array de nodos recorridos a *false* y lanza recursivamente una búsqueda en profundidad recorriendo las aristas salientes de peso *es* en el primer nivel de recursión y *subClaseDe* en los siguientes.

- *Tamaño de la entrada:* longitud de sujeto ≡ l, número de nodos ≡ n, número de aristas ≡ a
- *Operación elemental:* asignación
- *Número de operaciones elementales en el caso peor:* Se recorren todos los nodos y todas las aristas añadiendo el resultado a la lista enlazada, cuatro asignaciones por nodo: 4n + a. La longitud del sujeto se puede considerar constante en el contexto de la práctica, luego:

$$O(n+a)$$

7b) Colección de las clases que son superclase de una clase:

```
public ListaEnlazada<String> superClasesDe(String clase) { ... }
```

Procedimiento similar al anterior, pero esta vez recorriendo únicamente las aristas salientes de peso *subClaseDe*.

- *Tamaño de la entrada:* longitud de sujeto ≡ l, número de nodos ≡ n, número de aristas ≡ a
- *Operación elemental:* asignación
- *Número de operaciones elementales en el caso peor:* Las mismas que en la búsqueda en profundidad anterior. La longitud del sujeto se puede considerar constante en el contexto de la práctica, luego:

$$O(n+a)$$

8) Colección de entidades que son de una determinada clase:

```
public ListaEnlazada<String> entidadesDeClase(String clase) { ... }
```

Una búsqueda en profundidad más, pero esta vez recorriendo las aristas entrantes al nodo que se pasa como parámetro, en lugar de las salientes. Si el peso es *subClaseDe*, se relanza la búsqueda desde el nodo origen de la arista, y si el peso es *es*, se añade el nodo origen al resultado.

- *Tamaño de la entrada:* longitud de sujeto ≡ l, número de nodos ≡ n, número de aristas ≡ a
- *Operación elemental:* asignación
- *Número de operaciones elementales en el caso peor:* Las mismas que en la búsqueda en profundidad anterior. La longitud del sujeto se puede considerar constante en el contexto de la práctica, luego:

$$O(n+a)$$

10) Colección de estudiantes distintos que cursan alguna asignatura de la que es encargado un determinado profesor:

```
public ListaEnlazada<String> estudiantesDelProfesor(String profesor) { ... }
```

Comienza igual que los otros métodos, pero la búsqueda se limita a una profundidad de 2, por lo que no es necesaria la recursión: recorre las aristas salientes de peso *encargadoDe*, y de cada nodo accedido, recorre las aristas entrantes de peso *cursa*.

- *Tamaño de la entrada:* longitud de sujeto ≡ l, número de nodos ≡ n, número de aristas ≡ a
- *Operación elemental:* asignación
- *Número de operaciones elementales en el caso peor:* Las mismas que en la búsqueda en profundidad anterior. La longitud del sujeto se puede considerar constante en el contexto de la práctica, luego:

$$O(n+a)$$

11) Colección de profesores distintos que trabajan para algún departamento de una universidad:

```
public ListaEnlazada<String> profesoresDeUniversidad(String universidad) { ... }
```

Igual que en el método 10, no es necesario usar recursión. En primer lugar se recorren las aristas entrantes a la universidad de peso *departamentoDe*, y de cada nodo de departamento, se recorren sus aristas entrantes en busca de las de peso *trabajaPara*.

- *Tamaño de la entrada:* longitud de sujeto ≡ l, número de nodos ≡ n, número de aristas ≡ a
- *Operación elemental:* asignación
- *Número de operaciones elementales en el caso peor:* Las mismas que en la búsqueda en profundidad anterior. La longitud del sujeto se puede considerar constante en el contexto de la práctica, luego:

$$O(n+a)$$

Adicionalmente, se definen un par de métodos para las operaciones de carga y descarga del almacén desde/a fichero de texto.

El método 9a carga un almacén desde fichero, que es la funcionalidad que hemos implementado en la constructora de almacén, por lo que lo único que hace es llamarla.

El método 9b escribe todas las sentencias en un fichero de texto. Necesita recorrer todos los nodos y todas las aristas del grafo, en el orden en el que se han ido insertando. Por lo tanto, la complejidad del método es **O(n+a)**, teniendo en cuenta que escribir una sentencia en un fichero se realiza en tiempo constante.

## 5.3 Tiempos de ejecución

Los tiempos de ejecución para los nuevos métodos implementados son los siguientes:

| | **Clases de un sujeto** | **Superclases de una clase** | **Entidades de una clase[1]** | **Estudiantes distintos de un profesor[2]** | **Profesores distintos de una universidad[3]** | **Descargar sentencias en un fichero** |
|---|---|---|---|---|---|---|
| **A0.txt** | 0.02 ms | 0.016665 ms | 0.012757 ms | 0.023064 ms | 0.014826 ms | 2.841882 ms |
| **A1.txt** | 0.006283 ms | 0.006857 ms | 0.017125 ms | 0.000459 ms | 0.000344 ms | 1.787224 ms |
| **A2.txt** | 0.086125 ms | 0.088616 ms | 0.096431 ms | 0.117465 ms | 0.236999 ms | 18.408197 ms |
| **A3.txt** | 0.007662 ms | 0.006091 ms | 0.013332 ms | 0.007509 ms | 0.01406 ms | 117.669424 ms |
| **A4.txt** | 0.00816 ms | 0.0077 ms | 0.015171 ms | 0.009348 ms | 0.016819 ms | 204.648148 ms |
| **A5.txt** | 0.015324 ms | 0.013447 ms | 0.020037 ms | 0.015708 ms | 0.022872 ms | 428.508859 ms |
| **A6.txt** | 0.000459 ms | 0.000383 ms | 0.000574 ms | 0.032986 ms | 0.043637 ms | 1084.006733 ms |

[1]Clase: <http://www.w3.org/2002/07/owl#ObjectProperty>
[2]Profesor: <http://www.Department0.University0.edu/FullProfessor0>
[3]Universidad: <http://www.University0.edu>

El parámetro sujeto y el contexto en el que se realizaron las pruebas es el mismo que en las versiones anteriores.

# 6. Actas de las reuniones

***28 de septiembre***

Este día nos hemos reunido por primera vez en grupo para acordar como vamos a abordar el proyecto de EDA. Hemos discutido las diferentes ideas que han surgido entre los participantes del grupo y hemos decidimos recoger las ideas principales y desarrollarlas con más profundidad. Tras varias deliberaciones, hemos llegamos a una idea inicial de la estructura que vamos a implementar para realizar el proyecto. Esta avanzada estructura nos permitirá realizar las funciones deseadas en menor tiempo y con más eficacia, aunque el coste de implementarlas será mayor.

***1 de octubre***

En esta segunda reunión hemos decidido varios aspectos a tener en cuenta y nos hemos repartido la tarea en varias partes:

- Hemos terminado de debatir las estructuras que vamos a utilizar en el proyecto.
- A cada miembro del grupo se le ha asignado un tipo de tarea a realizar:
    - Asier realizará la implementación del árbol necesario para la carga del fichero y construcción del grafo.
    - Iván realizará las funciones respecto a la lectura/escritura del fichero.
    - Daniel probará casos de prueba para las funciones creadas anteriormente por Asier e Iván, y realizará la parte de la memoria correspondiente a los primeros pasos del proyecto.

Conclusión del acta: Tras un día de trabajo en clase, se han completado con éxito todas las expectativas. En la siguiente reunión grupal se decidirán los nuevos trabajos a cada miembro del grupo.

***3 de octubre***

En esta tercera reunión hemos decidido algunas nuevas ideas

para modificar y mejorar el proyecto:

- La estructura para hacer los ejercicios que se de decidió en la primera reunión ha resultado ser inválida. Por lo que hemos decidido una nueva manera de implementación, tal y como se refleja detalladamente en el apartado *Versión 0* de la memoria.

***5 de octubre***

Considerando los puntos a mejorar resaltados durante la primera evaluación con el profesor, en esta cuarta reunión hemos decidido repartir el trabajo a cada miembro del grupo rotando los papeles para que todos participemos por igual:

- Iván aclarará el código del proyecto para una mejor comprensión de las estructuras y hacer el código más legible.
- Daniel realizará algunos cambios en los algoritmos para adaptarlos al enunciado del proyecto. También realizará el análisis de complejidad de algunos de los métodos implementados.
- Asier efectuará los pertinentes cambios a la memoria acorde al seguimiento del proyecto y culminará el análisis de los algoritmos.

***14 de octubre***

Los 4 primeros apartados del proyecto se dan por finalizados aunque aún estén sujetos a modificaciones. Comenzamos a pensar cómo dar forma e implementar los algoritmos necesarios para los siguientes métodos, entre los que se incluyen la intersección de almacenes y ordenación. Acordamos que cada uno lleve una idea al próximo laboratorio.

***19 de octubre***

El laboratorio programado se aprovecha para debatir los puntos a seguir. Se acuerda la implementación de los métodos 5 y 6, y se pacta una nueva forma de distribuir la tarea: a medida que se escribe código en Java, la misma persona que programa añade su explicación a la memoria, en lugar de dejar que unos escriban código y otro actualice la

memoria. De esta forma se reparte mejor la carga de trabajo entre todos.

Así pues, Daniel adaptará las estructuras a los nuevos métodos planeados, Asier implementará el apartado 5, e Iván el número 6.

***25 de octubre***

Ponemos en común el trabajo realizado desde la última reunión. Las tareas de Daniel e Iván se encuentran terminadas y explicadas en su apartado de la memoria, no así la tarea de Asier que no ha tenido tiempo suficiente para hacerlo.

Se le permite a Asier tomarse algo más de tiempo en el método 5 dada su dificultad, y Daniel e Iván, a su vez, acuerdan observar el enunciado para continuar pensando o implementando métodos que crean sencillos. También se decide aprovechar el tiempo extra para repasar la corrección de la memoria.

***3 de noviembre***

Surgen diversos acontecimientos que retrasan el desarrollo normal del proyecto. Por una parte, la continuación de Daniel en el grupo pende de un hilo, por lo que se suspenden temporalmente sus actividades. A dos semanas de la competición SWERC en la que tomarán parte Asier e Iván, el tiempo del que disponen para trabajar en el proyecto se reduce drásticamente. En conclusión, se decide paralizar la actividad del grupo durante al menos una semana y media.

***15 de noviembre***

Tras hablar con el profesor y decidir lo que es más conveniente para él, Daniel nos comunica su salida definitiva del grupo. Asier e Iván seguirán al frente del proyecto como grupo de dos componentes. Queda una semana para la segunda fecha de evaluación del proyecto, y 2 días para el SWERC. Se acuerda continuar con el proyecto a la vuelta de la competición, disponiendo del tiempo justo para terminar el método 5 y actualizar la memoria.

***23 de noviembre***

En vistas a la cantidad de trabajo que se acumula para las últimas semanas lectivas, acordamos dar un empujón final a la práctica para terminarla cuanto antes y tener tiempo para otras asignaturas. No se prevé demasiada dificultad para los métodos que quedan por implementar.

El reparto de tareas es equitativo: métodos 7 y 10 para Iván, 8 y 11 para Asier. Cada uno añade a la memoria la explicación de sus métodos.

***1 de diciembre***

En este momento queda muy poco para terminar. Iván ha concluido la implementación que le correspondía y a Asier le falta muy poco. Iván se ofrece para dar un repaso al último apartado de la memoria y añadir algún punto que haya quedado sin comentar. Asier deberá añadir los tiempos de ejecución de los últimos métodos.

En esta reunión surgen otras ideas para optimizar el rendimiento: usar arrays estáticos en todos los métodos, aplicar técnicas de programación concurrente para las ordenaciones… Estas ideas quedan pendientes de llevar a cabo en función del tiempo y motivación de la que se disponga, a falta de conocer el enunciado extra que propondrá el profesor. Todo ello iría en una hipotética Versión 3.

# 7. Anexos

### *Anexo versión 0*
*Código fuente de lo implementado para la versión 0:*
*https://github.com/Sulley38/ProyectoEDA/raw/master/Version0.zip*

### *Anexo versión 1*
*Código fuente de lo implementado para la versión 1.*
*https://github.com/Sulley38/ProyectoEDA/raw/master/Version1.zip*

### *Anexo versión 2*
*Código fuente de lo implementado para la versión 2.*
*https://github.com/Sulley38/ProyectoEDA/raw/master/Version2.zip*