# Memoria WordCounter

A la hora de realizar la práctica, la primera decisión que se ha tomado es acerca de que estructuras de datos se han de usar para S1 y S2. En primer lugar, S1, s1 debe de ser una estructura de datos que nos permita hacer inserciones rápidamente, puesto que su principal cometido es agrupar una serie de objetos que contienen una palabra como clave, así como un registro del número de palabras, además en el caso de que una palabra esté repetida, nuestra estructura de datos permanecerá inalterada, puesto que tan solo se modificará el valor del objeto cuya clave coincida con la palabra ya existente. Por ello elegimos un HashMap frente a otro tipo de estructuras de datos. Los mapas nos permiten un rápido y fácil acceso a una clave que ya haya sido introducida, sin necesidad de ir comparando uno a uno los valores de los diferentes objetos que ya está introducidos en la estructura. Esta es la ventaja principal que nos plantean los diccionarios, por ello la complejidad de los métodos que impliquen añadir datos, será mucho menor que si se usa otro tipo de estructuras, como podrían ser los arboles binarios, que nos darían una complejidad de O(nlogn) para el caso de añadir, mientras que nuestra elección nos permita una complejidad de O(1) o en el peor de los casos O(n). La mayor contra que puede presentar el uso de estructuras con Hashes es que su ordenación es sumamente complicada, sin embargo, como se nos pide el uso de una estructura de datos paralela que actúa como buffer en el que se vuelcan los datos para su ordenación y procesado a través del resto de métodos programados en la práctica. El hecho de que esta estructura de datos sea un array, facilita bastante su ordenación, además de que esta es un tanto especial, puesto que el Objeto que se almacena en dicho array, tiene su propia implementación del método “compare to” adaptado a las necesidades de la práctica. Esta estructura de datos, s2, podría ser una lista, pero es más tediosa de ordenar, incluso según la implementación, podría no ser posible. Sin embargo, hemos pensado que podría ser factible el uso de un SET, puesto que no puede haber objetos repetidos en la estructura, y según la implementación que se emplee de un SET, la ordenación puede ser automática. La mayor contra que tendría haber empleado esta estructura de datos, es la dificultad adicional que plantea el volcado de datos desde el HashMap al SET, mientras que, con el array, viene ya implementado.

## Complejidad de los métodos

La complejidad del método put radica en una llamada al método get de la clase HashMap, la cual tiene complejidad constante O(1) al tratarse de una estructura de datos basada en hashes.

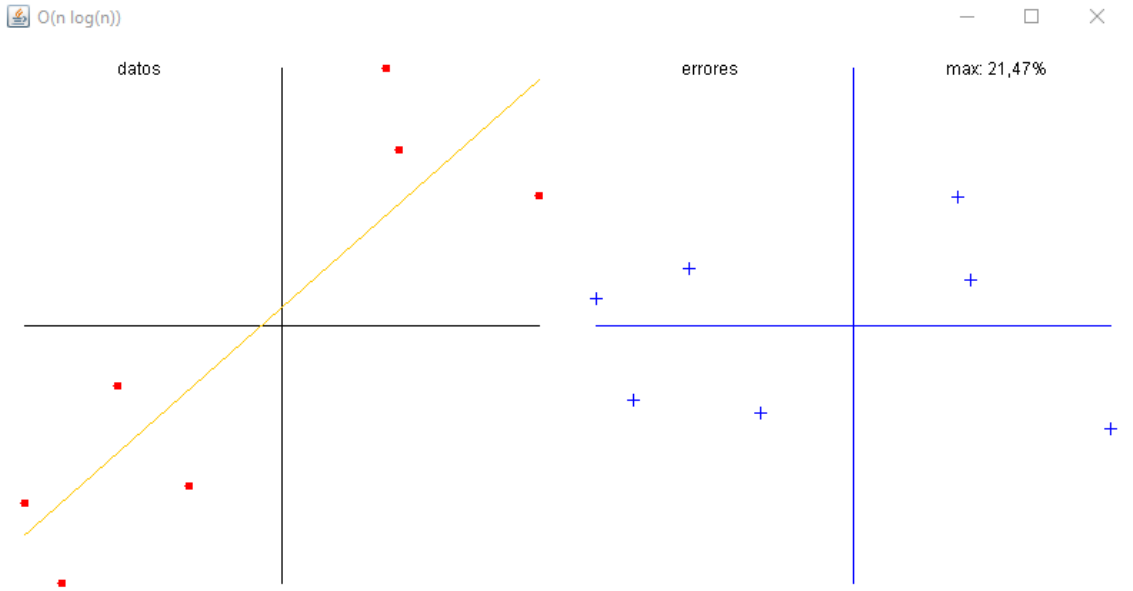
Para calcular la complejidad de getTop, procedemos a hacer la siguiente consideración, en los casos normales, para los que está pensado el método, ocurre que la complejidad del bucle for en getPalabrasMasUsadas y getPalabrasMenosUsadas es despreciable, cuando en el peor de los casos, mostrar todo el contenido de la estructura, aportaría O(n). Con esta consideración tenemos que tener en cuenta la complejidad de getDatos, la cual será por un lado la del proceso de volcado de datos de s1 a s2 y la ordenación de s2 con una función propia de java. El volcado de datos se hace con un bucle for que depende del número de datos por lo que aporta complejidad O(n). La complejidad de Arrays.sort() la obtenemos de la documentación java. Según esta es O(log(n)). La complejidad total de este método en casos normales será O(nLog(n)),

mientras que en el peor de los casos, donde no podemos despreciar la complejidad de getPalabrasMenosUsadas y su análogo, será de O($n^2$)

Para medir la complejidad empíricamente hemos usado los siguientes libros y hemos promediado los tiempos de ejecución a partir de 10 ejecuciones consecutivas del programa

| Libro | Numero de Palabras | Tiempo de Ejecución |
|---|---|---|
| **Lazarillo de Tormes** | 4395 | 144 |
| **Drácula** | 9515 | 287 |
| **Huckleberry** | 6484 | 123 |
| **Los Miserables** | 22930 | 410 |
| **Don Quijote** | 23591 | 502 |
| **Drácula y Lazarillo** | 13196 | 147 |
| **Lazarillo, Huckleberry y Don Quijote** | 30213 | 347 |

Usando la herramienta Correlator podemos verificar que la complejidad deducida anteriormente es correcta, puesto que es la mejor aproximación a los datos obtenidos en la tabla anterior.



## Test Realizados

Para comprobar el correcto funcionamiento de los métodos implementados hemos hecho los siguientes test:

- MasMenosUsadosTest: Prueba el correcto funcionamiento de getTop con índices negativos e índices positivos, llegando a los casos limites donde se toma toda la estructura
- RepetidosTest: Verifica que funciona correctamente la inserción de Strings con palabras repetidas
- VacioTest: Verifica el correcto funcionamiento de put y getTop en el caso límite de recibir un string vacío
- VacioTestFile: Idéntico al anterior solo que leyendo un .txt vacío
- EstándardTest: Prueba el funcionamiento usual de toda la clase
- NoAsciiTest: Prueba el comportamiento de wordcounter con caracteres no ASCII
- SimbolosTest: Prueba usando un caso marginal con muchos símbolos si la clase identifica de forma correcta las palabras
- DashTest: Prueba que las palabras separadas por guiones se cuentan correctamente.

## Libros Adicionales Empleados

Drácula: http://www.gutenberg.org/ebooks/345

Los Miserables: http://www.gutenberg.org/ebooks/135