

ANALIZADOR DE FICHEROS DE TEXTO



16 FEBRERO

AUTOR:

Mario Ventura Burgos 43223476-J

PROGRAMACIÓN 1

Profesor: Miquel Mascaró Portells



INTRODUCCIÓN:

EXPLICACIÓN DEL OBJETIVO DE LA PRÁCTICA

La práctica consiste en la creación de un programa capaz de leer y analizar ficheros de texto codificados en UTF-8 (documento de texto sin formato predeterminado de Windows).

Se pretende que el programa finalizado sea capaz de buscar un fichero cuyo nombre introduce el usuario que está usando el programa; y en caso de existir dicho fichero de texto, el programa leerá el contenido y lo mostrará (Usando el método de lectura carácter a carácter) en pantalla, junto a un conjunto de informaciones.

Estas informaciones son: Cantidad de caracteres, cantidad de palabras y cantidad de líneas de texto que tiene el fichero.

Posteriormente, se muestra un menú donde el usuario puede escoger entre 8 opciones más para analizar el fichero. Estas opciones son: Mostrar la letra más repetida y su número de apariciones (en caso de empate mostrar solo una de ellas), mostrar el número de apariciones de cada carácter, mostrar la palabra más repetida a lo largo del texto y su número de apariciones, buscar una palabra concreta en el texto (mostrar fila y columna en las que se encuentra), buscar un texto (es decir, un conjunto de palabras como “érase una vez”) y mostrar su posición (línea y columna), buscar cuando aparecen dos palabras iguales (consecutivas), codificar y decodificar el fichero (mediante una generación aleatoria predeterminada en el programa o mediante un número “semilla” que actuará a modo de contraseña o de “generador” de una secuencia aleatoria concreta) en otro nuevo fichero llamado “Codificado.txt” que se encontrará en la misma ubicación que el fichero de texto original analizado por el programa, y por último, establecer dicha “semilla”.

Estas son las funciones con las que debe contar el programa, pero, además, en nuestro caso, hemos añadido un conjunto de informaciones adicionales como:

Cantidad de vocales, cantidad de consonantes, línea más larga del texto, palabra más larga del texto, etc...

A modo de entender mejor todas estas funciones del menú y cuando deben aparecer, se ha creado la siguiente tabla:

OPCION	MODO DE FUNCIONAMIENTO	APARECE:
1	Nº de letras del fichero de texto	SIEMPRE
2	Nº de palabras del fichero de texto	SIEMPRE
3	Nº de líneas del fichero de texto	SIEMPRE
4	Muestra la letra más repetida y su número de apariciones.	CON EL MENÚ
5	Muestra el número de apariciones de cada carácter	CON EL MENÚ
6	Muestra la palabra más repetida y su número de apariciones.	CON EL MENÚ
7	Busca una palabra.	CON EL MENÚ
8	Busca un texto	CON EL MENÚ
9	Busca las palabras repetidas.	CON EL MENÚ
10	Codifica el fichero.	CON EL MENÚ
11	Establece la semilla de decodificación para decodificar el fichero	CON EL MENÚ
12	Información extra	CON EL MENÚ

Siendo este el objetivo propuesto, veamos cuál es el planteamiento con el cual se ha conseguido resolver el ejercicio.

DISEÑO DESCENDENTE:

¿QUÉ HA CONDUCTIDO A LA SOLUCIÓN PROPUESTA?

En primer lugar, y antes de empezar a escribir y desarrollar código, debemos pensar:

¿Cuáles son las diferentes formas de resolver el ejercicio?, de todas las formas posibles, ¿cuál es la óptima y la que me generará menos errores? ¿Qué clases necesito?

Como se trata de implementar los conceptos asimilados de la POO (Programación Orientada al Objeto), lo óptimo será desarrollar una clase en la que se ejecute el programa principal, usando métodos creados en esa misma clase u otra (mayoritariamente estarán creados en otras). Lo primero que tenemos que pensar por tanto es: ¿Qué clases necesita el programa?

Bien, vamos a enumerar las clases necesarias e indispensables, y la razón por la que lo son. Comencemos por entender que se trata de un analizador de ficheros de texto, por lo tanto, efectivamente, necesitaremos la clase Palabra, ya que vamos a tener que analizar y tratar con objetos Palabra a lo largo de toda la práctica. Además, necesitaremos añadir la clase LT, que permitirá leer lo que introduce por teclado el usuario que está usando el programa. También se necesitarán las clases **FicheroIn** y **FicheroOut**, ya que en algunos de los modos de los que dispone el menú se necesita crear ficheros nuevos y grabar información en ellos. Por último, crearemos la clase en la que se ejecutarán los distintos métodos creados en ella u otras clases.

En nuestro caso, también hemos añadido una clase llamada “MÉTODOS”, que contiene todas las funciones del menú. Es una clase donde usaremos los métodos de las clases **FicheroIn**, **FicheroOut**, **Palabra** y **LT** para hacer funcionar todas las opciones del menú. Posteriormente, los métodos de la clase MÉTODOS serán invocados a su vez en el programa principal, es decir, la clase donde se ejecuta todo.

De esta forma, además de quedar todo mucho más limpio, claro y ordenado; la orientación al objeto nos asegura que, si aparece algún error, este será mucho más fácil de localizar y corregir. Por ejemplo, si aparece un error al contar palabras que aparecen de forma consecutiva en el texto, podemos ir directamente a buscar el error en el método dedicado a esa función, y optimizamos el tiempo de búsqueda del error al reducir considerablemente las líneas de código que pueden contener el error.

Estas son solo algunas de las ventajas que ofrece la POO.

Comencemos a crear el código y a resolver los problemas mediante la creación de métodos.

En la clase principal del programa (la compilable y ejecutable), se crea un único método que llamaremos **Principal()** y que contiene únicamente métodos definidos en la clase métodos. Ahí correrá nuestro programa, pero para ello, debemos crear los métodos en la clase métodos, por lo que, de momento, el método **Principal()** quedará vacío.

En la clase MÉTODOS, comenzaremos a crear los diferentes métodos que necesitemos. Empezando por orden, lo primero que necesitamos es el lector del fichero.

Es obligatorio en este caso concreto el uso del **Buffered Reader** (recordar que lee carácter a carácter mediante una estructura iterativa). Lo pondremos en nuestra clase MÉTODOS y le pondremos el nombre que queramos. En nuestro caso, **LeerFicheroYMas()**. Pasaremos como parámetro una variable String que será el nombre del fichero a leer. Esto se hará en absolutamente todos los métodos que necesiten leer el fichero, es decir, todos los métodos del menú, ya que, de esa forma, al pasar como parámetro un nombre de fichero ya definido en un principio, se tratará siempre sobre el mismo fichero sin necesidad de volver a pedir al usuario que introduzca el nombre.

En realidad, esto es más complicado de lo que parece ya que no podemos usar String de forma directa, por tanto, se debe explicar más en profundidad y hacer un método a parte que permita leer lo que el usuario introduce por teclado y lo convierta en String. Para ello, en resumidas cuentas, haremos lo siguiente: Usaremos la clase LT para que el usuario introduzca un array de caracteres (recordar que un array es una lista de una cantidad definida de variables de un mismo tipo que pueden tener diferentes valores) y posteriormente usaremos uno de los constructores de la clase Palabra (usaremos aquel que tiene un array de caracteres como paso de parámetro) para convertir este array introducido por el usuario en un objeto

Palabra. Por último, usaremos un método de la clase Palabra para convertir este objeto Palabra en String (método **toString()**), que ya viene predeterminado en Java, y de ahí la etiqueta de “Override” que aparece en el IDE) ya que el lector de ficheros necesita que el parámetro que se introduzca sea un String.

Si además tenemos en cuenta la posibilidad de que el usuario introduzca un fichero que no exista, o de que pulse el botón *Enter* sin haber introducido ningún carácter, u otro tipo de comportamiento que conducirían a nuestro programa a un error; el método resultante para obtener el nombre del fichero a analizar será el siguiente:

```
//MÉTODO PARA OBTENER DEL USUARIO EL NOMBRE DEL FICHERO A LEER
public Paraula NombreFichero() {
    LT tec = new LT();
    MÉTODOS m = new MÉTODOS();
    File fil = null;
    while (fil == null) {
        m.PedirNombre();
        NomFic = new Paraula(tec.llegirLiniaA());

        if (NomFic != null) {
            fil = new File(NomFic.toString());
            if (!fil.exists()) {
                System.out.println("\n\tERROR: FICHERO INEXISTENTE O NO ENCONTRADO\n\n");
                fil = null;
            }
        } else {
            System.out.println("\n\tERROR: FICHERO INEXISTENTE O NO ENCONTRADO\n\n");
            fil = null;
        }
    }
    return NomFic;
}
```

Método público con un objeto Palabra como tipo de retorno que preguntará de forma repetida por el nombre del fichero hasta que se introduzca uno válido

Ahora mismo, el programa ya sabe que fichero leer y también sabe leerlo. Sin embargo, no muestra ninguna de las informaciones que se esperan en un primer momento.

Será necesario ahora, añadir funciones a nuestro método de lectura de fichero. Empecemos por contar caracteres.

·CONTAR CARACTERES

Para ello, se ha diseñado un método a parte llamado **EsCaracter(char c)**, también dentro de la clase MÉTODOS, y con tipo de retorno int, que verifica que el carácter leído por el Buffered Reader sea un carácter que se debe contar. Estos caracteres se introducen en un array, y se va comparando lo que ha leído el Reader con cada componente del array. En caso de encontrar que ambos son iguales, quiere decir que se ha encontrado un carácter que se debe contar, por

lo que una variable de tipo número entero a la que llamamos *numcaracteres* (inicialmente tiene valor 0) aumenta en 1 su valor (*numcaracteres++*;

Por tanto, si llamaos “c” al carácter que lee el Reader en su estructura iterativa, y colocamos el método descrito previamente dentro, sencillamente con pasar como parámetro un carácter, lo que hará el lector, además de leer es analizar carácter a carácter el fichero de texto, y en caso de encontrar un carácter que se debe contar, contarlo.

Veamos ahora como contar líneas, que resulta ser algo muy sencillo.

·CONTAR LÍNEAS

Como en realidad un ordenador no entiende otro idioma que no sean las matemáticas y los números, lo que hace en realidad el lector es, en pocas palabras, asociar a cada carácter/signo un valor numérico (ejemplo tabla ascii), de ahí que en el “while” del lector en Buffered, la condición sea que se siga leyendo mientras no se encuentre con un -1.

¿Cómo nos ayuda todo esto a contar líneas? Muy simple. Como en Java el salto de línea se puede escribir “\n”, se crea una variable constante de tipo entero llamada SALTOLINEA cuyo valor es “\n”. Por tanto, contar líneas será tan simple como crear una variable entera que cuente el número de líneas, y añadir dentro del bucle del Reader la condición de que, si el carácter leído == SALTOLINEA, entonces la variable que cuenta el número de líneas aumenta.

En este caso, sin embargo, la variable que cuenta el número de líneas será iniciada con valor 1, ya que las líneas no las contamos desde el 0 (porque se asume que, si un fichero de texto existe, significa que mínimo contiene algo dentro, y por tanto hay una línea de texto)

·CONTAR PALABRAS

Para contar palabras, como una palabra es, en definitiva, un conjunto de caracteres/letras, y por tanto una entidad un tanto más compleja, se ha hecho de la siguiente forma:

Se crean 2 variables de tipo carácter: “letrap”, que se iniciará con valor ‘ ’ (espacio); y “letra”, a la que inicialmente no se le da valor (se declaran las variables fuera de la estructura iterativa). Posteriormente, dentro del bucle que contiene el lector, haremos que la variable “letra” prenda el valor del carácter leído (c). Para contar palabras lo que haremos será pensar en que si la letra precedente a la variable letra (letrap) es espacio, punto, coma, punto y coma, etc... pero la variable letra no lo es (la variable letra siempre es la letra posterior a letrap, lógicamente) se entenderá que se ha encontrado el inicio de una palabra, y, por tanto, nuevamente, se hará aumentar en 1 una variable de tipo entero que inicialmente tiene valor 0 y que actúa como contador de palabras.

Después, se hará que la variable letrap tenga el valor de la variable letra, y la variable letra, tendrá el valor del nuevo carácter leído. De esta forma, ambas variables van avanzando de letra en letra, analizando y contando todas las veces que se encuentran con el inicio de una palabra. Es por esto por lo que la variable letrap se inicia con el valor ' '. Para poder contar la primera palabra.

Una de las alternativas sería contar el final de cada palabra, pero esto llevaría a ciertos errores ya que el lector del fichero deja de leer en cuanto llega al final del texto, y habría siempre un mínimo de una palabra que no se contaría.

Cabe destacar también, que para poder hacer esto, el lector debe ser capaz de reconocer y diferenciar entre una letra del abecedario y un espacio, una coma, un punto, etc... Con intención de que esto sea así, crearemos el método **Separador(char c)**, que es un simple método con tipo de retorno booleano que devuelve "true" en caso de que el carácter leído (el parámetro) sea un separador de palabra. Se verá de la siguiente manera:

```
/*MÉTODO BOOLEAN QUE DEVUELVE TRUE AL IDENTIFICAR ALGUNO DE LOS SEPARADORES
DE PALABRAS MÁS COMUNES*/
private boolean Separador(char c) {
    return ((c == ' ') || (c == ',') || (c == '.') || (c == ';') || (c == ':')
        || (c == '-') || (c == '_') || (c == '@') || (c == '#') || (c == '?')
        || (c == '¿') || (c == '!') || (c == '!') || (c == '+') || (c == '/')
        || (c == SALTOLINEA));
}
```

Hecho esto, ya podemos leer el fichero de texto que nos introduzca el usuario y contar los caracteres, palabras y líneas que este tiene. Mostraremos el resultado en pantalla mediante *System.out.println(introducir las variables que hacen de contadores);*

Con esto concluye la primera parte del programa, que es aquella previa al menú.

Nuestros próximos objetivos serán, por orden, los de la tabla de la página 3. Procedemos por tanto a mostrar las letras más repetida

LETRA MÁS REPETIDA

Para poder comprobar cuál es la letra con una mayor frecuencia de aparición en el texto, primero es necesario saber cuántas veces aparece cada letra. Una vez sepamos eso, podremos mirar cuál de ellas aparece más.

Para resolver este apartado, por tanto, serán necesarios dos arrays de diferentes variables:

-**char alfabeto[]**: Array de caracteres que contiene todas las letras del abecedario

-int frecuencias = new int[alfabeto.length]: Array de números enteros que contiene la cantidad de veces que aparece cada letra.

Como cada letra del array alfabeto irá asociada a una posición en el array de frecuencias, el array de frecuencias deberá ser del mismo tamaño que el array de letras que se quieren contabilizar, ya que, de ser menor, habría una letra cuya frecuencia de aparición no se podría contar, y de ser mayor, sobrarían espacios en la lista de frecuencias. Es por esto por lo que el array de frecuencias tiene como tamaño **alfabeto.length**.

Llegados a este punto, con un pequeño algoritmo que veremos a continuación, podremos contar las apariciones de cada letra. Este algoritmo es el siguiente:

Como vemos, con un algoritmo de apenas 10 o 15 líneas, somos capaces de contar las veces que aparece cada letra a lo largo del fichero de texto, aplicando los esquemas de recorrido y búsqueda y usando estructuras iterativas.

```
char alfabeto[] = "abcdefghijklmnopqrstuvwxyz".toCharArray();
int frecuencias[] = new int[alfabeto.length];
while (valor != -1) {
    char c = (char) valor;

    //-----PROCESO-----//
    //CONTAR APARICIONES Y ANOTARLAS EN EL ARRAY FRECUENCIAS[]
    int idx = 0;
    if (c == alfabeto[idx]) {
        frecuencias[idx]++;
    } else {
        while ((idx < alfabeto.length) && (c != alfabeto[idx])) {
            idx++;
        }
        if (idx < alfabeto.length) {frecuencias[idx]++;}
    }
    valor = reader.read();
}
```

Explicuemos un poco más en profundidad y de una forma más gráfica, como se ha llegado a esta solución. Para ello, vamos a representar nuestros arrays de frecuencias y caracteres (con valores inventados):

Caracteres

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Frecuencias (valores de ejemplo)

3	6	7	1	0	5	5	6	0	9	2	4	2	15	9	6	9	3	2	0	1	4	4	3	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	----	---	---	---	---	---	---	---	---	---	---	---	---

Como vemos, se puede asociar a cada carácter, un valor del array de frecuencias, que será el de su misma posición. Por tanto, en este ejemplo concreto, la frecuencia de aparición de cada letra sería: la 'a' 3 veces, la 'b' 6 veces, la 'c' 7 veces, etc...

La única dificultad reside por tanto en ser capaz de hacer que cada vez que aparezca una de las letras del array de caracteres, se sume 1 en la “caja” del array de frecuencias de su misma posición. Para ello usamos una variable *idx* que inicialmente vale 0, y que mediante un bucle “**for**” va creciendo. Lo que hace este bucle es comparar la letra leída por el lector de fichero y si es igual al array de caracteres en posición *idx* (es decir, *alfabeto[0]*) se suma 1 al array de frecuencias en posición *idx* (*frecuencias[idx]*). De no ser así, se realizará la orden *idx++* y se volverá a comprobar si el valor leído por el Buffered Reader coincide con el de *alfabeto[idx]*, y así sucesivamente. Esto se repetirá hasta que se encuentre que el valor leído coincide con alguno de los de *alfabeto[]*, o bien hasta que ya se hayan comparado todas las letras del array alfabeto con el carácter leído, en cuyo caso no se suma nada al array de frecuencias ya que no se ha encontrado ninguna letra.

Una vez hecho esto, nuevamente mediante un bucle “**for**” se compararán todos los valores del array de frecuencias y se mostrará el mayor. Para hacer esto, la idea es muy simple.

Se declara una variable llamada posición de tipo número entero que se iniciará con valor 0. Posteriormente se declara una variable de tipo entero llamada mayor cuyo valor inicial también es 0, y se asume que, en un principio, la mayor = *alfabeto[0]*. Con un bucle for que esta vez empieza en posición 1 (int i=1;), se compara mediante un if el valor de la variable *mayor* con el valor de frecuencias[i]. Si la variable *mayor* es la mayor de ambas, está seguirá siendo la mayor de las frecuencias registradas, pero de lo contrario, la mayor de las frecuencias será *frecuencias[1]*.

Sea como sea, posteriormente se hace i++; y se compara la siguiente casilla del array de frecuencias con la variable mayor. De esta forma, al comparar todos los valores de la lista, obtenemos que la variable mayor contiene el mayor valor contenido en la lista de frecuencias, y que está en posición i (dentro del for, en caso de que salga que *mayor > frecuencias[i]* es falso, la posición de la mayor variable es la posición número i). Por esto es necesario definir la variable posición; ya que, al acabar la comparación de las frecuencias, la variable i se destruye, pero el de posición no, ya que ha sido declarada fuera del bucle.

Ahora tan solo deberemos mostrar los resultados en pantalla declarando que la letra que más veces se repite es *alfabeto[posición]* y que se repite un total de “*mayor*” veces.

Basándonos en este método, al que hemos llamado **LetraMasRepetida()**, podremos crear también el próximo método de la lista, que consiste en mostrar las apariciones de cada letra.

·APARICIONES DE CADA LETRA

Para ver cuantas veces aparece cada letra a lo largo del texto, se hará exactamente lo mismo que en el método anterior, excepto por la impresión en pantalla de los resultados.

Se creará un array con todas las letras y un array con las frecuencias, se contará cuantas veces aparece cada letra, etc... Lo único que cambia respecto al método anterior es que ahora ya no hay que comparar cuál de las frecuencias de aparición es la mayor. Ahora sencillamente las mostraremos todas.

¿Cómo? Nuevamente, con estructuras iterativas. La idea de recorrido y búsqueda es vital para resolver cada uno de los problemas que la práctica plantea, y por ello, si sabemos usarlos debidamente, podremos resolver dichos problemas de forma eficaz.

En esta ocasión, en vez de un “**for**”, por tal de cambiar respecto al método anterior, usaremos un **while** y una variable **i** declarada fuera del **while**, y que inicialmente tiene valor 0. Una vez tengamos las frecuencias de aparición de cada carácter, crearemos el siguiente bucle:

```
//MOSTRAR RESULTADOS
System.out.println("\tAL LEER EL FICHERO, SE HA OBTENIDO QUE: \n\n");
int i = 0;
while (i < alfabeto.length) {
    System.out.println("\tEL N° DE VECES QUE APARECE EL CARACTER '" + alfabeto[i] + "' ES DE " + frecuencias[i]);
    i++;
}
```

De esta forma se muestran todas las letras y su frecuencia de aparición.

```
EL N° DE VECES QUE APARECE EL CARACTER 'a' ES DE 18
EL N° DE VECES QUE APARECE EL CARACTER 'b' ES DE 5
EL N° DE VECES QUE APARECE EL CARACTER 'c' ES DE 9
EL N° DE VECES QUE APARECE EL CARACTER 'd' ES DE 13
EL N° DE VECES QUE APARECE EL CARACTER 'e' ES DE 34
EL N° DE VECES QUE APARECE EL CARACTER 'f' ES DE 3
EL N° DE VECES QUE APARECE EL CARACTER 'g' ES DE 0
EL N° DE VECES QUE APARECE EL CARACTER 'h' ES DE 6
EL N° DE VECES QUE APARECE EL CARACTER 'i' ES DE 12
EL N° DE VECES QUE APARECE EL CARACTER 'j' ES DE 0
EL N° DE VECES QUE APARECE EL CARACTER 'k' ES DE 0
```

·PALABRA QUE MÁS VECES APARECE EN EL FICHERO

Para encontrar la palabra cuya frecuencia de aparición es mayor o igual a la del resto, primero deberemos saber cuántas veces aparece cada palabra. Para ello, crearemos un array de números enteros que cuente la frecuencia de aparición de cada palabra. Así pues, como el número máximo de palabras en el fichero es de 500, no necesitaremos un array de más de 500 casillas. **Inciso: Para facilitar el posible cambio de este valor, se crea una constante entera llamada “MAXIMO” que tendrá valor 500 y podrá ser usada en toda la clase**

Lo crearemos por tanto con el tamaño de la variable “MAXIMO” (500) y comenzaremos a darle valor a cada “casilla”. ¿Cómo lograremos hacer esto?

El proceso es complejo, ya que tenemos que asociar a cada palabra un número entero que representa las apariciones a lo largo del fichero, pero no podemos almacenar objetos palabra. Haremos uso de un array llamado “letras1” de capacidad 35 para asegurarnos de que cabe todo tipo de palabra, y de otro llamado “letras2” cuyo tamaño se definirá luego.

Inciso: Como el valor 35 (capacidad de letras1) se usará varias veces a lo largo del programa, se crea una constante entera llamada CAPACIDAD cuyo valor es 35

En resumidas cuentas, el proceso llevado a cabo para llegar a la solución es el siguiente: Lo que se hará será leer la primera palabra del fichero y compararla con el resto de las palabras del fichero (y si dos palabras son iguales hacer que la frecuencia de aparición aumente en 1), leer la segunda palabra del fichero y compararla con el resto de las palabras, leer la tercera, la cuarta, y así sucesivamente hasta haber leído y comparado todas las palabras con todas.

Esto plantea un problema ya que para leer una palabra y compararla con el resto, debemos almacenarla, pero no podemos. Para ello, lo que en realidad haremos es lo siguiente:

- 1) Crear una variable entera llamada “posición” que inicialmente vale 0.
- 2) Hacer una primera lectura del fichero y guardar la primera palabra de este en el array “letras2”, que tiene el tamaño justo de la palabra leída, ya que en realidad es una copia del array “letras1” pero sin espacios sin usar (que posteriormente desaparece ya que no se pretende almacenar nada).
- 3) Mientras sigan quedando palabras por leer en la primera lectura, se vuelve a leer todo el fichero desde el principio con otro BufferedReader y a medida que se vaya leyendo, cada palabra que aparezca se compara con la palabra almacenada en “letras2”.
- 4) Si en esta segunda lectura del fichero, la palabra contenida en “letras2” es igual a la palabra leída, se aumenta en 1 el valor de frecuencias[posición] (si es la 1ª palabra del texto, posición = 0; si es la palabra nº 10, posición = 9, etc...), y si no, la frecuencia no cambia (sigue siendo 0 o el valor que ya tenía).
- 5) Eliminar la palabra contenida en “letras2”, hacer que la variable posición aumente en 1, y volver al punto 3.

Esto se repetirá mientras siga habiendo palabras por comparar, y nos dejará como resultado un array de frecuencias en el que cada casilla tiene asignado el valor del número de veces que aparece cada palabra. Esto se entiende mejor con un pequeño ejemplo visual:

CONTENIDO DEL FICHERO	LA	VIDA	ES	LA	COSA	MÁS	BONITA
APARICIÓN DE CADA PALABRA	2	1	1	2	1	1	1

Es necesario comprender que en realidad el programa, siguiendo el ejemplo anterior, solo tendría almacenados los números, las palabras no. Esto es debido a que como hemos mencionado anteriormente, el programa no almacena palabras, solo va asignando números a un array de frecuencias en función del valor de una variable posición.

También es necesario saber que para comparar 2 arrays se ha creado un método llamado ***SonIgualesArray()***, con tipo de retorno boolean, que es el siguiente:

```
//MÉTODO BOOLEAN PARA COMPARAR DOS ARRAYS DE CARACTERES
private boolean SonIgualesArray(char[] a, char[] b) {
    int x = 0;
    if (a.length == b.length) {
        for (int i = 0; i < a.length; i++) {
            if (a[i] == b[i]) {
                x++;
            }
        }
        if (x == a.length) {
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
}
```

Método boolean que devuelve true cuando dos arrays de caracteres pasados por parámetro son iguales, es decir, cuando tienen la misma longitud (mismo número de casillas) y tienen el mismo contenido en cada una de las casillas de misma posición que forman estos arrays.

Llegados a este punto lo que en realidad tenemos es una lista de números que en realidad no están asociados a nada. Por esto mismo, para mostrar los resultados haremos lo siguiente:

- 1) Para saber que, valor del array de frecuencias es el mayor de todos (la palabra más repetida a lo largo del texto) haremos el mismo proceso hecho anteriormente para ver la letra más repetida. Crearemos la variable “mayor” y haremos el proceso siguiente:

```
//AHORA YA SABEMOS CUANTAS VECES APARECE CADA PALABRA, POR TANTO PROCEDEMOS COMPARAR PARA VER CUÁL APARECE MÁS
int mayor = frecuencias[0];
int pos = 0;
for (int i = 1; i < frecuencias.length; i++) {
    if (frecuencias[i] > mayor) {
        mayor = frecuencias[i];
        pos = i;
    }
}
```

- 2) Crear una nueva variable “*pos*” que representa la posición de las palabras en el fichero.
 - 3) Leer por tercera vez el fichero, registrando cada palabra y haciendo que cada palabra que se lea tenga un valor de posición. Si el array de frecuencias en la casilla [*pos*] es igual a el valor de la variable *mayor*, quiere decir que hay varias palabras que aparecen un mismo número de veces (La palabra más repetida aparece X número de veces, pero hay otras palabras que también aparecen X número de veces).
- Es decir, se lee el fichero hasta encontrar la primera palabra, que al ser la primera tiene un valor de *pos* = 0. Si *frecuencias[pos] == mayor*, mostrar en pantalla esta palabra.
- 4) Aumentar en 1 el valor de la variable *pos* y leer la palabra siguiente para volver a hacer lo mismo.

De esta forma logramos mostrar por pantalla la palabra que más se repite a lo largo del fichero, y en caso de empate se muestran todas ellas.

A continuación, un pequeño fragmento del código usado y el resultado final:

Código para mostrar los resultados

```
//LEER FICHERO UNA VEZ MÁS PARA PODER MOSTRAR LA PALABRA EN CONCRETO, YA QUE NO SE PUEDEN ALMACENAR PALABRAS
while (valor3 != -1) {
    char r = (char) valor3;
    if (Separador(r) == false) {
        letras5[ind] = r;
        ind++;
    } else {
        if (r == SALTOLINEA) {
            letras6 = new char[ind - 1];
        } else {
            letras6 = new char[ind];
        }
        for (int i = 0; i < letras6.length; i++) {
            letras6[i] = letras5[i];
        }

        //SI LA PALABRA TIENE UNA FRECUENCIA DE APARICION == A "mayor", MOSTRARLA TAMBIÉN
        if ((frecuencias[lugar] == mayor) && (letras6.length >= 1 /*ES DECIR, QUE letras6 NO ESTÁ VACÍO*/)) {
            System.out.print("\tLA PALABRA \t");
            for (int x = 0; x < letras6.length; x++) {
                System.out.print(letras6[x]);
            }
            System.out.print("\t \tAPARECE UN TOTAL DE " + frecuencias[lugar] + " VECES A LO LARGO DEL FICHERO\n");
        }
    }
}
```

Resultado final con el texto “prueba1.txt”

```
-----LA PALABRA/PALABRAS MÁS REPETIDA/REPETIDAS EN EL TEXTO SON-----

LA PALABRA      'la'      APARECE UN TOTAL DE 4 VECES A LO LARGO DEL FICHERO
LA PALABRA      'es'      APARECE UN TOTAL DE 4 VECES A LO LARGO DEL FICHERO
```

BUSCAR UNA PALABRA EN EL TEXTO:

Para buscar una palabra en el texto, el proceso será mucho más simple.

Para este proceso, únicamente necesitaremos 2 arrays: *letras1* con capacidad de 35 letras para asegurarnos de que cabe todo tipo de palabra, y *letras2*, cuya función veremos más adelante.

Lo que se hará es pedir al usuario que introduzca la palabra que quiere buscar, y una vez hecho eso, procederemos a leer el fichero y buscarla en él. Para poder buscar la palabra introducida dentro del contenido del fichero, obviamente necesitaremos leer el fichero, asique el proceso será el siguiente:

Se irán leyendo caracteres, y mientras estos no sean separadores (cualquiera de los comprendidos en el método **Separador**), se irán añadiendo a *letras1*. En el momento en el que se encuentre un separador, lo que se hará es definir el tamaño de *letras2*, dándole un tamaño exacto del número de casillas necesarias para que se pueda copiar en él todo el contenido de *letras1* pero que no sobren casillas vacías.

Hecho esto, tenemos nuestra primera palabra del fichero lista. Mediante el método **SonIgualesArray**, la comparamos con la palabra introducida por el usuario, y si son iguales (cuando el método devuelve true) entonces se imprime la posición de esta palabra. Si no son iguales obviamente no se imprime la posición.

Hecho esto, se lee la próxima palabra del fichero y se vuelve a hacer lo mismo, y así sucesivamente hasta que ya no queden palabra por comparar.

Así, conseguimos mostrar por pantalla la ubicación (o las distintas ubicaciones/apariciones) en el texto de la palabra que ha introducido el usuario.

Para mostrar la ubicación de una palabra, el proceso siempre es el mismo: se crea una variable que es un contador de columnas, otra que es un contador de líneas, y dos arrays: uno de líneas y otro de columnas.

Cada vez que se lee un carácter, si este no es un salto de línea, el número de columnas aumenta. Si es un salto de línea entonces vuelve a su valor inicial, que es 1.

Las líneas se inician con valor 1 y ese valor aumenta cada vez que el carácter leído es un salto de línea.

De esta forma, cada vez que se lee un carácter, este tiene un valor de línea y de columna que muestra su ubicación. Estos valores son los que mostraremos cuando encontremos la palabra dentro del fichero de texto. Cada vez que un método requiera mostrar la ubicación de una palabra o un texto, se hará de esta manera

Si no se ha encontrado la palabra en el fichero, también se debe informar al usuario, por tanto se hará lo mismo que en el método anterior: Si después de leer todo el fichero, la variable *NoEncontrado* == 0, entonces se informa al usuario de que no se ha encontrado la palabra (esto implica que se tendrá que crear la variable con valor inicial de 0, y que cada vez que se encuentre la palabra el valor de NoEncontrado deberá crecer, para que así no salte el aviso luego cuando la palabra sí que ha sido encontrada).

Además, el hecho de comparar palabra a palabra hace que, si el usuario en vez de una palabra introduce un texto, este nunca será encontrado, ya que no es la función de este método

```
-----¿QUE PALABRA DESEAS BUSCAR?-----
-----
ESCRIBE UNA PALABRA A BUSCAR -----> 1234
SE HA ENCONTRADO LA PALABRA EN LA LÍNEA      1      Y EN LA COLUMNA 8
SE HA ENCONTRADO LA PALABRA EN LA LÍNEA      2      Y EN LA COLUMNA 14
SE HA ENCONTRADO LA PALABRA EN LA LÍNEA      3      Y EN LA COLUMNA 8
```

BUSCAR UN TEXTO:

Para buscar un texto, pese a que este es una entidad más compleja que una palabra (ya que un texto es un conjunto de palabras separadas entre sí con un espacio o cualquier signo de puntuación), la solución será mucho más simple

Este proceso será casi inmediato, sin necesidad de crear nuevos arrays cuyo tamaño será adaptado posteriormente, ni ninguna de las “ayudas” o “ideas” que hemos hecho anteriormente.

La idea es la siguiente: Se le pide al usuario que introduzca un texto que quiera buscar, y una vez más, lo que se lee es un array de caracteres al que llamaremos *usuario[]*. Posteriormente, crearemos 2 variables enteras llamadas “*encontrado*” y “*NoEncontrado*” cuyo valor inicial es 0; tal y como hemos hecho al buscar palabras o al buscar la palabra más repetida, y haremos lo siguiente:

Se leerá el fichero letra a letra (como siempre) y se comparará con lo que ha introducido el usuario. Si se encuentra que la letra leída es igual a *usuario[encontrado]*, entonces la variable *encontrado* crece (*encontrado++*). De esta forma, al leer la próxima letra, esta se comparará con la siguiente letra, es decir, *usuario[encontrado]*, donde *encontrado* ahora valdrá 1.

Por el contrario, si en algún momento la letra que se lee no coincide con la letra de *usuario[encontrado]*, la variable *encontrado* volverá a valer 0.

Si en algún momento la variable *“encontrado”* tiene el mismo valor que *usuario.length*; es decir, si se ha encontrado una cadena de caracteres en el fichero que coincide con la que se ha introducido por teclado, entonces se muestra por pantalla la línea y la columna en la que se está trabajando, y se informa al usuario de que se ha encontrado el texto allí. Posteriormente la variable *“encontrado”* vuelve a tener valor 0 y se sigue leyendo hasta que no queden más caracteres por leer en el fichero.

El resultado que todo este proceso nos mostrará en pantalla es el siguiente:

```
-----
-----EL FICHERO CONTIENE EL SIGUIENTE TEXTO:-----
-----
la vida es es es una maravilla sobre todo cuando estudias en la uib, y mas aun si estudias una una ingenieria
Visca la vida i tot el que es
la vida y el mundo son grandes, bonitos y muy muy curiosos

-----
-----INTRODUCE EL TEXTO QUE QUIERAS BUSCAR-----
-----
ESCRIBE UN TEXTO A BUSCAR -----> si estudias una una ingenieria
SE HA ENCONTRADO EL TEXTO EN LA LÍNEA   1          COLUMNA       109
```

BUSCAR PALABRAS REPETIDAS:

El proceso con el cual encontraremos si una palabra está repetida en el texto es relativamente simple. Lo que haremos será leer el fichero e ir almacenando palabras de 2 en 2 y compararlas para que, en caso de que sean iguales, indicar que se encuentran repetidas y mostrar su posición (mostrar línea y columna tal y como se ha hecho anteriormente).

Para hacer esto, una vez más haremos el mismo proceso que se hizo anteriormente con los arrays *“letras1”* y *“letras2”*, pero además esta vez usaremos un nuevo array llamado *“letras3”*. Las funciones de estos arrays son:

Letras1: Array de caracteres con capacidad *CAPACIDAD* (35) que servirá para que a medida que se lea el fichero carácter a carácter, en el momento en el que acaba una palabra, esta queda almacenada de forma temporal en *letras1*. Al tener tamaño 35, muy probablemente sobrarán espacios vacíos en el array. Para eso mismo está el array *letras2*

Letras2: Array con el tamaño justo y necesario para contener la palabra almacenada en *letras1* sin que sobre ningún espacio vacío; es decir, es un array que contiene la palabra registrada en *letras1*

Letras3: Copia de *letras2*. Array con la misma capacidad y mismo contenido (se copia *letras2* en *letras3*)

Ahora que ya sabemos que, hace cada array, veamos como estos nos pueden ayudar a ver que palabras aparecen de forma consecutiva en el fichero de texto. El proceso es el siguiente:

- 1) Se leerá el fichero, se almacenará la primera palabra en *letras1* y se copiará a *letras2* con el tamaño justo. Posteriormente se crea el array *letras3* con el mismo tamaño que *letras2*, pero aún no contiene nada.
- 2) Se compara el contenido de *letras2* con el de *letras3* (la 1ª vez obviamente el método ***SonIgualesArray()*** devolverá false ya que el array *letras3* está vacío de momento).
- 3) Se copia el contenido de *letras2* en *letras3* y se lee la siguiente palabra del fichero, que se almacena en *letras2*.
- 4) Se vuelve al punto 2

De esta forma se compara siempre la primera palabra con la segunda, la segunda con la tercera, la tercera con la cuarta, etc... Cada una de las palabras del fichero será comparada con la palabra que tiene delante, y en caso de ser iguales se notifica al usuario.

De esta forma conseguimos ver que palabras aparecen repetidas de forma consecutiva en el texto contenido en el fichero de texto.

```
NOMBRE DEL FICHERO CON EL QUE SE ESTÁ TRABAJANDO: prueba1.txt

1) VISUALIZAR LA LETRA MÁS REPETIDA A LO LARGO DEL FICHERO
2) VISUALIZAR LAS APARICIONES DE CADA LETRA A LO LARGO DEL FICHERO
3) VISUALIZAR LA PALABRA QUE MÁS VECES APARECE EN EL FICHERO
4) BUSCAR UNA PALABRA EN EL FICHERO
5) BUSCAR UN TEXTO EN EL FICHERO
6) BUSCAR TODAS LAS PALABRAS QUE APARECEN REPETIDAS (DE FORMA CONSECUTIVA)
7) CODIFICAR FICHERO DE TEXTO
8) ESTABLECER UNA SEMILLA DE CODIFICACIÓN DEL FICHERO
9) VER OTRA INFORMACIÓN (INFORMACIÓN EXTRA SOBRE EL FICHERO)
10) PARA SALIR DEL MENÚ, ESCRIBE '10' Y PULSA ENTER

NÚMERO DE LA ORDEN A REALIZAR -----> 6

SE HA ENCONTRADO LA PALABRA 'es' REPETIDA EN LA LÍNEA 1 COLUMNA 13
SE HA ENCONTRADO LA PALABRA 'es' REPETIDA EN LA LÍNEA 1 COLUMNA 16
SE HA ENCONTRADO LA PALABRA 'una' REPETIDA EN LA LÍNEA 1 COLUMNA 98
SE HA ENCONTRADO LA PALABRA 'muy' REPETIDA EN LA LÍNEA 3 COLUMNA 50
```

CODIFICAR EL FICHERO (Y DECODIFICAR)

Para codificar el contenido del fichero en otro fichero nuevo, en primer lugar, se tendrá que introducir una semilla de codificación, por tanto, antes de codificarlo se usará un método por el cual el usuario deberá introducir dicha semilla. Veamos estos dos métodos por separado para ver cómo funciona cada uno de forma individual:

•**Para codificar el fichero de texto**, tendremos que convertir cada carácter en uno aleatorio, basado en la semilla que se introduzca. Según el número que introduzca el usuario (que debe ser entre 0 y 9999) se generará una codificación en concreto u otra; es decir, hay 10.000 posibles codificaciones diferentes en función del número que el usuario escoja.

Esto lo lograremos de forma que se pase el valor de la semilla como parámetro del método. Hecho esto, lo que el método hace es crear un nuevo fichero de texto llamado Codificado.txt cuya ubicación es la misma que la del fichero original. El programa leerá carácter a carácter el fichero de texto, y cada carácter que lea un carácter lo sustituirá por otro en el fichero Codificado.txt.

Esto es posible gracias un método llamado **ProcesarArchivo()**, que tiene como parámetros dos arrays de caracteres.

El primero de ellos, es una lista de todos los caracteres que se deben codificar, y el segundo es un array de codificación generado en función del valor de la semilla introducida por el usuario.

Lo que hace este método, en pocas palabras, es convertir cada carácter del primer array que se le pasa por parámetro, en uno de los caracteres del segundo array que tiene como parámetro y gravarlo en un fichero nuevo.

Tabla de ejemplo de una de las posibles codificaciones que se pueden generar

a→p	g→?	m→d	s→m	y→g	?→r	>→{
b→c	h→h	n→!	t→a	z→o	!→s	
c→.	i→"	o→)	u→e	.→:	"→u	
d→k	j→<	p→t	v→@	,→n	(→b	
e→l	k→f	q→y	w→i	:→>)→j	
f→,	l→v	r→w	x→q	@→z	<→x	

Explicuemos de forma más detallada lo que hace el método **ProcesarArchivo()**.

Mediante el uso de las clases FicheroIn y FicheroOut, lee el fichero de texto original, crea el nuevo fichero llamado Codificado.txt y en función del valor de la semilla, si el carácter leído en

el fichero original es un valor que se debe codificar, lo hace, y si no, lo deja como está (no codifica mayúsculas, por ejemplo).

Cada carácter que se lee pasa por este proceso, y es gravado en el nuevo fichero.

El resultado final será que gracias a la creación de dos arrays (el original es siempre el mismo, el codificado cambia en función de la semilla) y este proceso de lectura, sustitución y escritura de caracteres; obtendremos un fichero de texto, cuyo contenido, casi con total seguridad, no tendrá ningún tipo de sentido desde el punto de vista del lenguaje humano.

Por tanto, dentro del método de codificar fichero (de la clase métodos) lo que haremos será invocar a este método ***ProcesarArchivo()***, que por tanto debe ser público, y pasar por parámetro los arrays original y codificado (en ese mismo orden), para que se pueda convertir cada carácter a su codificación concreta en caso de que sea preciso.

Si lo que queremos es decodificar el fichero codificado previamente, como lo que hace el método es convertir cada carácter del primer array que tiene como parámetro en uno del array que tiene como segundo parámetro; lo que haremos será invocar de nuevo el método ***ProcesarArchivo()***, pero esta vez, los parámetros irán en orden inverso. Es decir, para codificar, como se debe convertir cada carácter original en uno codificado, el método es el siguiente: ***ProcesarArchivo(original, codificado)***. En caso de que queramos decodificar el fichero, el proceso es el inverso: convertir el fichero codificado al original, por tanto, se hará ***ProcesarArchivo(codificado, original)***. Así cada carácter codificado volverá a su estado original, pero el resultado se verá en el texto Codificado.txt, que por tanto debe tener exactamente el mismo contenido que el fichero analizado originalmente.

Hecho esto, se cambiará el valor de la variable *NomFic*, que ahora será "Codificado.txt". Expliquemos esto último:

Todo nuestro programa funciona de forma en que cada uno de los métodos que requieren de leer el fichero de texto, tienen como parámetro una variable String que es el nombre de ese fichero. De esta forma, como es la misma variable la que se pasa por parámetro de todos estos métodos, se lee y se trabaja siempre sobre el mismo fichero. Así pues, como el objetivo del programa es que una vez se haya codificado el texto, se trabaje con él de forma en que los resultados sean los mismos que si no se hubiese codificado (cosa que indica que la codificación ha salido bien), lo que haremos será cambiar el valor de la variable *NomFic*.

Si recordamos bien, lo primero que hace el programa al ejecutarse, es pedir al usuario que introduzca el nombre del fichero que quiere leer, y este se almacena en la variable *NomFic*.

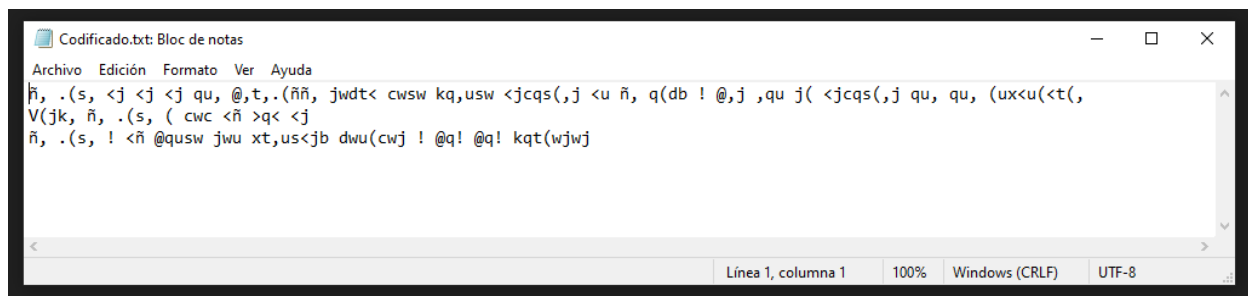
De esta forma, cada vez que se realiza un método, como este tiene como parámetro este nombre de fichero, siempre se trabaja sobre ese; pero esto cambia en el momento en el que se codifica el fichero.

Como ahora queremos trabajar sobre el fichero codificado lo que haremos será cambiar el valor de la variable *NomFic*, que como ya hemos dicho ahora valdrá “Codificado.txt”.

Así al seguir usando el programa, cada uno de los métodos está trabajando realmente sobre el fichero codificado.

Pese a esto, si la codificación está bien hecha, los resultados tienen que ser los mismos, es decir, si mientras estamos trabajando sobre el fichero codificado quiero mirar el número de veces que se repite cada letra a lo largo del fichero, el resultado debe ser el mismo que si se trabajase sobre el fichero sin codificar.

EJEMPLO DE TEXTO CODIFICADO



EJEMPLO DE COMO EL FUNCIONAMIENTO ES EL MISMO

```
NOMBRE DEL FICHERO CON EL QUE SE ESTÁ TRABAJANDO: Codificado.txt
SEMILLA DE CODIFICACIÓN: 6789

1)      VISUALIZAR LA LETRA MÁS REPETIDA A LO LARGO DEL FICHERO
2)      VISUALIZAR LAS APARICIONES DE CADA LETRA A LO LARGO DEL FICHERO
3)      VISUALIZAR LA PALABRA QUE MÁS VECES APARECE EN EL FICHERO
4)      BUSCAR UNA PALABRA EN EL FICHERO
5)      BUSCAR UN TEXTO EN EL FICHERO
6)      BUSCAR TODAS LAS PALABRAS QUE APARECEN REPETIDAS (DE FORMA CONSECUTIVA)
7)      CODIFICAR FICHERO DE TEXTO
8)      ESTABLECER UNA SEMILLA DE CODIFICACIÓN DEL FICHERO
9)      VER OTRA INFORMACIÓN (INFORMACIÓN EXTRA SOBRE EL FICHERO)
10)     PARA SALIR DEL MENÚ, ESCRIBE '10' Y PULSA ENTER

NÚMERO DE LA ORDEN A REALIZAR -----> 10

AL LEER EL FICHERO, SE HA OBTENIDO QUE:

LA LETRA MÁS REPETIDA ES LA LETRA 'a'
QUE SE REPITE UN TOTAL DE 21 VECES
```

Pero para que todo esto funcione el usuario tiene que introducir una semilla. Veamos cómo se hace esto.

PEDIR SEMILLA:

Para que el usuario introduzca una semilla, el proceso es muy simple. Se crea un método con tipo de retorno int y con una simple estructura iterativa que solo acepta números entre 0 y 9999 se consigue esta semilla.

Esta estructura iterativa funciona de forma en que introduce una variable llamada res cuyo valor inicial es -1, y posteriormente dice que, mientras esta variable valga -1, se va a pedir al usuario que introduzca un número entre 0 y 9999. Si el número no está en este rango, el valor de la variable res volverá a ser -1, y por tanto se volverá a entrar en este bucle, del que no se saldrá hasta que no se introduzca un número dentro del rango.

Una vez el usuario haya introducido un número que sea admitido (es decir, un número de entre 0 y 9999), el método devolverá este valor, que pasará a ser almacenado como variable global que puede ser usada en todo el programa.

Así se obtiene la semilla de codificación.

```
//-----ESTABLECER "SEMILLA" DE CODIFICACIÓN-----
Integer res = -1;
private int CogerSemilla() {
    LT tec = new LT();
    System.out.println("\t-----");
    System.out.println("\t----INTRODUCE UNA SEMILLA DE CODIFICACIÓN (Nº ENTRE 0 Y 9999)-----");
    System.out.println("\t-----ESTA SEMILLA GENERARÁ UNA CODIFICACIÓN EN CONCRETO-----");
    System.out.println("\t-----\n\n");
    while (res <= -1) {
        res = tec.llegirSencer();
        if ((res == null) || (res <= -1) || (res > 10000)) {
            System.out.println("\t-----ERROR-----\n\n");
            res = -1;
        }
    }
    return res;
}
```

Pero ¿Por qué se genera una codificación u otra en función de un valor numérico?

Como ya se ha mencionado con anterioridad, para llevar a cabo la codificación del fichero, se debe pasar por parámetro del método ProcesarArchivo() un array que sea el valor de codificación de cada carácter contenido en el array original.

Se pueden generar diferentes codificaciones en función del valor de la semilla porque, pese a que el array original nunca cambia, el array de codificación sí. Cambia en función del valor de la semilla, ya que este array está creado mediante una secuencia pseudoaleatoria.

Los ordenadores son solo máquinas que siguen órdenes humanas, hacen lo que se les pide con gran eficacia, pero, al fin y al cabo, no son creativos. No son capaces de crear ni de pensar, únicamente de seguir las órdenes que se le dan. Solo son capaces de seguir procesos algorítmicos con gran velocidad, hacer cálculos y procesos con números y variables dadas, etc...

Si se le pide a un ordenador que genere un número aleatorio, en realidad el número que se genere no es aleatorio, ya que eso implicaría una capacidad de decisión y de pensamiento sin seguir ningún tipo de norma. Un pensamiento libre que no atiende a ninguna fórmula o algoritmo.

Por eso, al generar números aleatorios, lo que los ordenadores hacen en realidad es generar un número en función de un conjunto de muchas variables que están cometidas a constante cambio, y que por tanto hacen parecer a este número, un número aleatorio. Si nosotros conociésemos estas variables y su valor en cada momento, podríamos predecir que, número se generará, ya que este no es aleatorio.

De la misma forma, también se puede forzar a que la generación aleatoria sea una en concreto de entre todas las posibilidades. Esta es la función de la semilla.

Se marca un límite de 10.000 codificaciones diferentes, y lo que se hace es generar un array de codificación que no será aleatorio, si no que dependerá del valor de la semilla. De esta forma nosotros sabemos en todo momento sobre cuál de las 10.000 posibilidades estamos tratando.

Para hacer esto, deberemos importar en la clase MÉTODOS el "Java.util.Random" y generar una secuencia en concreto de la siguiente manera:

```
Random ran = new Random(semilla)
```

De esta forma se genera una "aleatoriedad controlada" a partir de la cual se genera el array de codificación.

ASPECTOS ADICIONALES:

Además de todas las funciones con las que el menú cuenta, se ha añadido una función extra en la que se reflejan algunos aspectos o características sobre el fichero que no se han expresado en otros métodos, por tal de mostrar al usuario más información sobre el fichero de texto. Algunos de estos aspectos son el número de letras, el número de vocales, el número de consonantes, etc...

Veamos cuales son estas informaciones adicionales y como se han podido deducir:

- 1) **Nº DE LETRAS:** Para contar el número de letras que aparecen en el texto (letras, no caracteres), se ha creado un método llamado ***EsLetra()*** que como parámetro incluye una variable *char*. Este método contiene un array con todas las letras del abecedario en mayúscula y minúscula, y lo que hace es comparar el valor de la variable pasada por parámetro con los valores contenidos en el array. En caso de que la letra sea igual que alguno de los valores, y, por tanto, sea una letra, el método devuelve true, ya que es un método con tipo de retorno booleano.

Como se lee el fichero carácter a carácter, cada carácter leído se pasará como parámetro de este método y por tanto se sabrá si es una letra o no. Cuando el método devuelve true, es decir, cuando se encuentra una letra, un contador llamado *numLetras* (cuyo valor inicial es 0) crece y suma 1.

- 2) **Nº DE VOCALES:** El proceso es el mismo que el anterior. Un método booleano que devuelve true si la letra que se le pasa por parámetro es vocal. Por tanto, este método contendrá un array con todas las vocales, en mayúsculas, minúsculas, con acento, sin acento, con diéresis y sin diéresis.

Cuando se encuentra que el carácter leído es una vocal un contador llamado *numVocales*, que inicialmente valía 0, aumenta en 1 su valor.

- 3) **Nº DE CONSONANTES:** Para contar el número de consonantes no hace falta ningún método, ya que lo único que hay que hacer es restar al número de letras el número de vocales; ya que todo lo que no sea vocal, si es letra, será consonante.
- 4) **Nº DE MAYÚSCULAS:** Es la misma idea que los contadores anteriores. Un método con tipo de retorno booleano que contiene un array con todas las letras del abecedario en mayúscula y que devuelve true si una letra que se le pasa por parámetro está contenida en ese array (o en otras palabras, si es mayúscula), en cuyo caso un contador irá aumentando
- 5) **Nº DE MINÚSCULAS:** Se obtiene haciendo la resta (*NumLetras - NumMayusculas*), ya que, si se lee una letra y no es mayúscula, entonces es minúscula.

6) **CANTIDAD DE NÚMEROS:** Para contar la cantidad de números que aparecen en el texto se hace lo mismo. Se crea un método boolean que contiene un array con todos los números del 0 al 9 y que pasa como parámetro una variable *char*, que será comparada con los números contenidos en el array. Si se encuentra que el carácter es un número un contador aumentará en 1 su valor.

7) **Nº DE PALABRAS CON 5 O MÁS VOCALES:** Para saber si una palabra tiene más de 5 vocales, se hace lo siguiente: A medida que se vayan leyendo caracteres se irán almacenando en un array *letras1* con capacidad para 35 caracteres. Una vez se detecte un separador, se crea un array *letras2* con el tamaño justo para contener la palabra almacenada en *letras1* sin que sobren ni falten espacios.

Una vez tenemos la palabra en *letras2*, usando el método ***EsVocal()***, lo que haremos será pasar cada letra contenida en *letras2* como parámetro del método de forma que, si este detecta que es una vocal, se suma 1 a un contador. Si al acabar de hacer esto con toda la palabra el contador es mayor o igual a 5, significa que tiene 5 vocales o más, y eso hará que el método ***Tiene5Vocales()*** devuelva true, ya que este es un método boolean.

Esto se hará con todas las palabras del texto de forma que, si este método devuelve true, un contador que cuenta el número de palabras con 5 vocales o más, aumentará.

8) **PALABRA MÁS LARGA DEL FICHERO:** Es un proceso que crea un array de número enteros llamado *longitudes[]* con una capacidad de 500, ya que este es el número máximo de palabras que admite el fichero.

Sobre cada palabra que se almacena en *letras2*, se guardará *letras2.length* en la posición respectiva del array (es decir, en posición 0 del array, la longitud de la primera palabra; en la posición 1, la segunda palabra, etc...).

Una vez se haya almacenado la longitud de todas las palabras, se hará un proceso para ver cual de los valores almacenados en *longitudes[]* es el mayor, es decir, cuál es la longitud de la palabra más larga, y la longitud de esta se guardará en una variable llamada mayor.

Este será el resultado que se muestre al final. Se informará al usuario de que la palabra más larga tiene un total de “mayor” caracteres/letras.

9) **Nº DE DIÉRESIS:** Es el mismo proceso que para contar vocales, letras, etc... Un método boolean que devuelve true o false en función si el carácter leído tiene diéresis o no. En caso de que tenga, un contador aumenta.

10) **Nº DE SIGNOS DE PUNTUACIÓN:** Mismo proceso, pero con signos de puntuación.

RESULTADOS QUE OFRECE ESTE MÉTODO SOBRE EL FICHERO prueba1.txt

```
-----INFORMACIÓN EXTRA SOBRE EL FICHERO-----  
  
1) EL FICHERO CONTIENE UN TOTAL DE 155 LETRAS  
2) DE ESAS LETRAS, 74 SON VOCALES Y 81 SON CONSONANTES  
3) DE ESAS VOCALES, HAY UN TOTAL DE 0 VOCALES CON ACENTO  
4) DE TODAS LAS LETRAS CONTENIDAS EN EL TEXTO, 1 SON MAYÚSCULAS Y 154 SON MINÚSCULAS  
5) DE TODOS LOS CARACTERES DEL TEXTO, 0 SON NÚMEROS  
6) ADEMÁS, HAY UN TOTAL DE 1 PALABRAS QUE CONTIENEN 5 O MÁS VOCALES  
7) LA PALABRA MÁS LARGA DEL TEXTO TIENE UNA LONGITUD DE 10 LETRAS  
8) SE HA ENCONTRADO UN TOTAL DE 0 LETRAS CON DIÉRESIS  
9) SE HA ENCONTRADO UN TOTAL DE 2 SIGNOS DE PUNTUACIÓN
```

MENÚ:

Hecho todo esto, ya tenemos todas las funciones de nuestro programa creadas, con lo cual solo nos falta crear el menú y ejecutar todo lo anterior.

El menú funcionará de forma en que se declararán tantas variables de tipo número entero como opciones haya, y se pedirá al usuario que desea hacer. El usuario deberá introducir el número de la orden a realizar (saldrán enumeradas para que el usuario sepa lo que está haciendo, obviamente).

En caso de que el usuario haya introducido un número menor o mayor a las posibilidades que el menú contempla, este mostrará un error y volverá a pedir al usuario un número a introducir. Lo mismo pasará si el usuario introduce algo que no sea un número, si se introduce un número con decimales, o si se pulsa la tecla enter sin haber introducido nada por teclado (null).

Cuando el usuario introduzca un número válido, se realizará la orden del número introducido (nº1 orden 1, nº2 orden 2, etc...) ya que cada orden tiene asociado un número, que es el del valor de la variable entera.

Todo esto sucederá mientras el usuario no introduzca el número 0, que tal y como informa el menú, sirve para salir del programa. En caso de que se introduzca este número, el programa se despedirá del usuario y terminará con su ejecución.

Cabe destacar un detalle más, y es que el programa no ejecutará el menú si el fichero supera las 500 palabras. Para que esto sea así, hemos creado un método que es igual que el método **LeerFicheroYMas()**, pero que no cuenta caracteres, ni líneas, ni muestra el contenido del

fichero. Solo cuenta el número de palabras que el fichero contiene, y en caso de superar las 500, no ejecuta el menú, si no que muestra un aviso al usuario y le informa de que el fichero es demasiado grande.

Este método para contar palabras podrá ser encontrado en la clase MÉTODOS y se llama ***ExcedeMaximo()***; y obviamente se ejecuta después de pedir al usuario que fichero desea leer, ya que de lo contrario no se podrían contar las palabras.

EJEMPLO DEL AVISO QUE MUESTRA EL PROGRAMA

```
-----INTRODUCE EL NOMBRE DEL FICHERO A LEER (DEBE ACABAR EN .TXT)-----  
-----EJEMPLO: TEXTO.TXT-----  
  
quijote.txt  
><><><><><><><>_____ERROR_____<><><><><><><><>  
---EL FICHERO EXCEDE EL MAXIMO DE PALABRAS. NO SE PUEDE PROCESAR---
```

IMAGEN DE LA EJECUCIÓN DEL MENÚ

JUEGO DE PRUEBAS:

A lo largo del informe ya se han mostrado algunos ejemplos con los que el programa funciona, pero para asegurarnos de que esto es así siempre, pongamos a prueba el programa con más ejemplos. Para estos ejemplos estaremos usando el fichero: prueba1.txt que contiene el siguiente texto:

“la vida es es es una maravilla sobre todo cuando estudias en la uib, y mas aun si estudias una una ingenieria

Visca la vida i tot el que es

la vida y el mundo son grandes, bonitos y muy muy curiosos”

El texto en sí no tiene demasiada coherencia, pero contiene lo suficiente como para probar que nuestro programa funciona, ya que contiene palabras repetidas, saltos de líneas, etc...

Comencemos a probar si el funcionamiento es el correcto:

```
NOMBRE DEL FICHERO -----> prueba1.txt

-----EL FICHERO CONTIENE EL SIGUIENTE TEXTO:-----
la vida es es es una maravilla sobre todo cuando estudias en la uib, y mas aun si estudias una una ingenieria
Visca la vida i tot el que es
la vida y el mundo son grandes, bonitos y muy muy curiosos

1) EL FICHERO CONTIENE UN TOTAL DE 157 CARACTERES,
2) EL FICHERO CONTIENE UN TOTAL DE 42 PALABRAS.
3) EL FICHERO CONTIENE UN TOTAL DE 3 LÍNEAS DE TEXTO
```

Se puede ver que el programa es capaz de leer el fichero y contar de forma correcta el número de palabras, caracteres y líneas que contiene el fichero de texto.

Acto seguido se abre el menú y se muestran las siguientes opciones:

```
-----
-----¿QUE DESEAS HACER CON EL FICHERO?-----
----- ESCRIBE EL NÚMERO DE LA ORDEN A REALIZAR -----
-----

NOMBRE DEL FICHERO CON EL QUE SE ESTÁ TRABAJANDO: prueba1.txt

1)      VISUALIZAR LA LETRA MÁS REPETIDA A LO LARGO DEL FICHERO
2)      VISUALIZAR LAS APARICIONES DE CADA LETRA A LO LARGO DEL FICHERO
3)      VISUALIZAR LA PALABRA QUE MÁS VECES APARECE EN EL FICHERO
4)      BUSCAR UNA PALABRA EN EL FICHERO
5)      BUSCAR UN TEXTO EN EL FICHERO
6)      BUSCAR TODAS LAS PALABRAS QUE APARECEN REPETIDAS (DE FORMA CONSECUTIVA)
7)      CODIFICAR FICHERO DE TEXTO
8)      ESTABLECER UNA SEMILLA DE CODIFICACIÓN DEL FICHERO
9)      VER OTRA INFORMACIÓN (INFORMACIÓN EXTRA SOBRE EL FICHERO)

PARA SALIR DEL MENÚ, ESCRIBE '0' Y PULSA ENTER

NÚMERO DE LA ORDEN A REALIZAR ----->
```

Probaremos cada una para ver si son correctas:

1). LETRA MÁS REPETIDA

El texto ha sido capaz de contar la cantidad de veces que aparece cada letra y después comparar todas esas apariciones para concluir cuál de ellas es mayor.

```
NÚMERO DE LA ORDEN A REALIZAR -----> 1

AL LEER EL FICHERO, SE HA OBTENIDO QUE:

LA LETRA MÁS REPETIDA ES LA LETRA 'a'
QUE SE REPITE UN TOTAL DE 21 VECES
```

Continuemos con la comprobación:

2). APARICIONES DE CADA LETRA

```
EL N° DE VECES QUE APARECE EL CARACTER 'a' ES DE 21
EL N° DE VECES QUE APARECE EL CARACTER 'b' ES DE 3
EL N° DE VECES QUE APARECE EL CARACTER 'c' ES DE 3
EL N° DE VECES QUE APARECE EL CARACTER 'd' ES DE 9
EL N° DE VECES QUE APARECE EL CARACTER 'e' ES DE 14
EL N° DE VECES QUE APARECE EL CARACTER 'f' ES DE 0
EL N° DE VECES QUE APARECE EL CARACTER 'g' ES DE 2
EL N° DE VECES QUE APARECE EL CARACTER 'h' ES DE 0
EL N° DE VECES QUE APARECE EL CARACTER 'i' ES DE 15
EL N° DE VECES QUE APARECE EL CARACTER 'j' ES DE 0
EL N° DE VECES QUE APARECE EL CARACTER 'k' ES DE 0
EL N° DE VECES QUE APARECE EL CARACTER 'l' ES DE 8
EL N° DE VECES QUE APARECE EL CARACTER 'm' ES DE 5
EL N° DE VECES QUE APARECE EL CARACTER 'n' ES DE 12
EL N° DE VECES QUE APARECE EL CARACTER 'ñ' ES DE 0
EL N° DE VECES QUE APARECE EL CARACTER 'o' ES DE 11
EL N° DE VECES QUE APARECE EL CARACTER 'p' ES DE 0
EL N° DE VECES QUE APARECE EL CARACTER 'q' ES DE 1
EL N° DE VECES QUE APARECE EL CARACTER 'r' ES DE 5
EL N° DE VECES QUE APARECE EL CARACTER 's' ES DE 17
EL N° DE VECES QUE APARECE EL CARACTER 't' ES DE 6
EL N° DE VECES QUE APARECE EL CARACTER 'u' ES DE 13
EL N° DE VECES QUE APARECE EL CARACTER 'v' ES DE 4
EL N° DE VECES QUE APARECE EL CARACTER 'w' ES DE 0
EL N° DE VECES QUE APARECE EL CARACTER 'x' ES DE 0
EL N° DE VECES QUE APARECE EL CARACTER 'y' ES DE 5
EL N° DE VECES QUE APARECE EL CARACTER 'z' ES DE 0
```

Como podemos ver, el programa cuenta exitosamente el número de veces que aparece cada letra.

Además, esto nos sirve para comprobar que, efectivamente, la letra 'a' es la que más veces aparece a lo largo del fichero

3). PALABRA QUE MÁS APARECE (MÁS REPETIDA)

```
-----LA PALABRA/PALABRAS MÁS REPETIDA/REPETIDAS EN EL TEXTO SON-----
LA PALABRA      'la'      APARECE UN TOTAL DE 4 VECES A LO LARGO DEL FICHERO
LA PALABRA      'es'      APARECE UN TOTAL DE 4 VECES A LO LARGO DEL FICHERO
LA PALABRA      'es'      APARECE UN TOTAL DE 4 VECES A LO LARGO DEL FICHERO
LA PALABRA      'es'      APARECE UN TOTAL DE 4 VECES A LO LARGO DEL FICHERO
LA PALABRA      'es'      APARECE UN TOTAL DE 4 VECES A LO LARGO DEL FICHERO
```

Una vez más, hemos conseguido lo que pretendíamos. El programa es capaz de mostrar con eficacia la palabra que más se repite en el fichero de texto, y en caso de empate, muestra todas (en este caso, las palabras 'es' y 'una').

4). BUSCAR PALABRA EN EL FICHERO

Al buscar una palabra, vemos que también el resultado el esperado. En el apartado anterior hemos visto que la palabra 'es' aparece un total de 4 veces, y ahora, al buscarla, vemos las 4 posiciones en el fichero de texto. Esto nos sirve para comprobar que, efectivamente, ambos métodos funcionan bien.

```
NÚMERO DE LA ORDEN A REALIZAR -----> 4
-----
-----¿QUE PALABRA DESEAS BUSCAR?-----
-----
ESCRIBE UNA PALABRA A BUSCAR -----> es
SE HA ENCONTRADO LA PALABRA EN LA LÍNEA      1      Y EN LA COLUMNA 11
SE HA ENCONTRADO LA PALABRA EN LA LÍNEA      1      Y EN LA COLUMNA 14
SE HA ENCONTRADO LA PALABRA EN LA LÍNEA      1      Y EN LA COLUMNA 17
SE HA ENCONTRADO LA PALABRA EN LA LÍNEA      2      Y EN LA COLUMNA 30
```

5). BUSCAR TEXTO

Para comprobar el correcto funcionamiento del método dedicado a la búsqueda de un texto dentro del fichero, introduciremos, por ejemplo, un conjunto de palabras que se encuentra al final de la primera línea. Veamos los resultados:

```
NÚMERO DE LA ORDEN A REALIZAR -----> 5
-----
-----INTRODUCE EL TEXTO QUE QUIERAS BUSCAR-----
-----
ESCRIBE UN TEXTO A BUSCAR -----> y mas que el resultado sea
SE HA ENCONTRADO EL TEXTO EN LA LÍNEA      1      COLUMNA      94
```

Tal y como pretendíamos, el programa es capaz de encontrar el texto e indicar correctamente su posición. Al ser el final de la primera línea, la línea es 1 y la columna tiene un número ciertamente elevado.

6). PALABRAS REPETIDAS (APARECEN DE FORMA CONSECUTIVA)

Busquemos ahora las palabras que aparecen repetidas de forma consecutiva en el fichero. Seleccionemos por tanto la opción 6 del menú:

```
NÚMERO DE LA ORDEN A REALIZAR -----> 6

SE HA ENCONTRADO LA PALABRA 'es' REPETIDA EN LA LÍNEA 1 COLUMNA 13
SE HA ENCONTRADO LA PALABRA 'es' REPETIDA EN LA LÍNEA 1 COLUMNA 16
SE HA ENCONTRADO LA PALABRA 'una' REPETIDA EN LA LÍNEA 1 COLUMNA 98
SE HA ENCONTRADO LA PALABRA 'muy' REPETIDA EN LA LÍNEA 3 COLUMNA 50
```

De nuevo, hemos conseguido mostrar la información que queríamos, cosa que demuestra la eficacia del método y de la abstracción planteada.

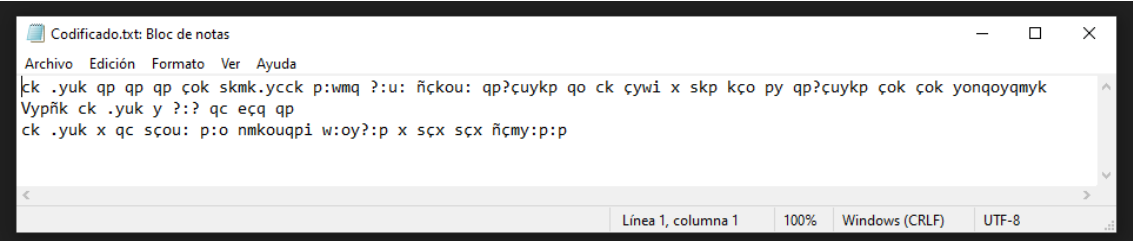
7). CODIFICAR TEXTO

Si se ha introducido una semilla previamente, se codificará el fichero con esa semilla; pero en caso de que el usuario se haya olvidado de introducir una semilla y haya seleccionado la opción de codificar el fichero, el programa lo detecta, y se la pide:

```
NÚMERO DE LA ORDEN A REALIZAR -----> 7

-----
----INTRODUCE UNA SEMILLA DE CODIFICACIÓN (Nº ENTRE 0 Y 9999)----
-----ESTA SEMILLA GENERARÁ UNA CODIFICACIÓN EN CONCRETO-----
-----
SEMILLA -----> 1234
```

Una vez se ha hecho esto, ya se puede codificar el fichero en base a esa semilla. Veamos el resultado de la codificación:



```
Codificado.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
ck .yuk qp qp qp cok skmk.ycck p:wmq ?:u: ñckou: qp?çuykp qo ck çywi x skp kço py qp?çuykp cok cok yonqoyqmyk
Vypñk ck .yuk y ??:? qc eçq qp
ck .yuk x qc sçou: p:o nmkouqpi w:oy?:p x sçx sçx ñçmy:p:p
Línea 1, columna 1 100% Windows (CRLF) UTF-8
```

Una vez codificado el fichero, antes de empezar a trabajar con él, se ofrece al usuario la posibilidad de decodificarlo.

```

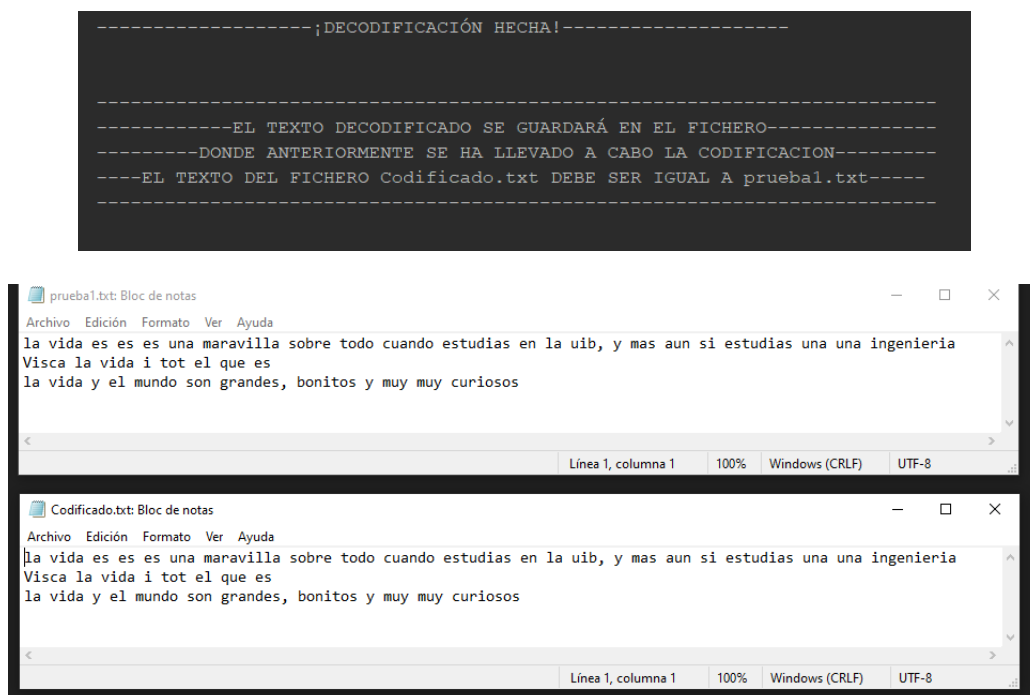
EL RESULTADO DE LA CODIFICACIÓN SE PODRÁ VER EN EL FICHERO 'Codificado.txt'

SI QUIERES SEGUIR TRABAJANDO CON EL FICHERO CODIFICADO ESCRIBE '1' Y PULSA ENTER
SI QUIERES DECODIFICAR EL FICHERO ANTES DE TRABAJAR CON ÉL, ESCRIBE '2' Y PULSA ENTER
INTRODUCE UNA DE LAS DOS OPCIONES ----->

```

Probemos ambas opciones:

7.1) DECODIFICAR ANTES DE TRABAJAR:



El resultado de la decodificación, tal y como vemos, ha sido el esperado, ya que tal y como informa el programa, el texto en el fichero codificado.txt es igual al del texto analizado (en este caso prueba1.txt). Ahora el programa pasará a trabajar con el fichero codificado.txt, tal y como él mismo nos informa:

```

NOMBRE DEL FICHERO CON EL QUE SE ESTÁ TRABAJANDO: Codificado.txt
SEMILLA DE CODIFICACIÓN: 1234

1) VISUALIZAR LA LETRA MÁS REPETIDA A LO LARGO DEL FICHERO
2) VISUALIZAR LAS APARICIONES DE CADA LETRA A LO LARGO DEL FICHERO
3) VISUALIZAR LA PALABRA QUE MÁS VECES APARECE EN EL FICHERO
4) BUSCAR UNA PALABRA EN EL FICHERO
5) BUSCAR UN TEXTO EN EL FICHERO
6) BUSCAR TODAS LAS PALABRAS QUE APARECEN REPETIDAS (DE FORMA CONSECUTIVA)
7) CODIFICAR FICHERO DE TEXTO
8) ESTABLECER UNA SEMILLA DE CODIFICACIÓN DEL FICHERO
9) VER OTRA INFORMACIÓN (INFORMACIÓN EXTRA SOBRE EL FICHERO)

PARA SALIR DEL MENÚ, ESCRIBE '0' Y PULSA ENTER

NÚMERO DE LA ORDEN A REALIZAR ----->

```

Probemos ahora a trabajar con el fichero codificado sin decodificarlo primero:

7.2) TRABAJAR CON EL FICHERO SIN DECODIFICARLO:

```
SI QUIERES SEGUIR TRABAJANDO CON EL FICHERO CODIFICADO ESCRIBE '1' Y PULSA ENTER
SI QUIERES DECODIFICAR EL FICHERO ANTES DE TRABAJAR CON ÉL, ESCRIBE '2' Y PULSA ENTER
INTRODUCE UNA DE LAS DOS OPCIONES -----> 1

DESEAS HACER ALGO MÁS?

NOMBRE DEL FICHERO CON EL QUE SE ESTÁ TRABAJANDO: Codificado.txt
SEMILLA DE CODIFICACIÓN: 1234

1) VISUALIZAR LA LETRA MÁS REPETIDA A LO LARGO DEL FICHERO
2) VISUALIZAR LAS APARICIONES DE CADA LETRA A LO LARGO DEL FICHERO
3) VISUALIZAR LA PALABRA QUE MÁS VECES APARECE EN EL FICHERO
4) BUSCAR UNA PALABRA EN EL FICHERO
5) BUSCAR UN TEXTO EN EL FICHERO
6) BUSCAR TODAS LAS PALABRAS QUE APARECEN REPETIDAS (DE FORMA CONSECUTIVA)
7) CODIFICAR FICHERO DE TEXTO
8) ESTABLECER UNA SEMILLA DE CODIFICACIÓN DEL FICHERO
9) VER OTRA INFORMACIÓN (INFORMACIÓN EXTRA SOBRE EL FICHERO)

PARA SALIR DEL MENÚ, ESCRIBE '0' Y PULSA ENTER

NÚMERO DE LA ORDEN A REALIZAR ----->
```

Como vemos, ahora se trabaja con el fichero codificado sin haberlo decodificado primero (opción 1).

Pese a esto, los resultados de las diferentes opciones del menú no deberían cambiar:

```
NOMBRE DEL FICHERO CON EL QUE SE ESTÁ TRABAJANDO: Codificado.txt
SEMILLA DE CODIFICACIÓN: 1234

1) VISUALIZAR LA LETRA MÁS REPETIDA A LO LARGO DEL FICHERO
2) VISUALIZAR LAS APARICIONES DE CADA LETRA A LO LARGO DEL FICHERO
3) VISUALIZAR LA PALABRA QUE MÁS VECES APARECE EN EL FICHERO
4) BUSCAR UNA PALABRA EN EL FICHERO
5) BUSCAR UN TEXTO EN EL FICHERO
6) BUSCAR TODAS LAS PALABRAS QUE APARECEN REPETIDAS (DE FORMA CONSECUTIVA)
7) CODIFICAR FICHERO DE TEXTO
8) ESTABLECER UNA SEMILLA DE CODIFICACIÓN DEL FICHERO
9) VER OTRA INFORMACIÓN (INFORMACIÓN EXTRA SOBRE EL FICHERO)

PARA SALIR DEL MENÚ, ESCRIBE '0' Y PULSA ENTER

NÚMERO DE LA ORDEN A REALIZAR -----> 4

-----
-----¿QUE PALABRA DESEAS BUSCAR?-----
-----

ESCRIBE UNA PALABRA A BUSCAR -----> vida
SE HA ENCONTRADO LA PALABRA EN LA LÍNEA      1      Y EN LA COLUMNA 8
SE HA ENCONTRADO LA PALABRA EN LA LÍNEA      2      Y EN LA COLUMNA 14
SE HA ENCONTRADO LA PALABRA EN LA LÍNEA      3      Y EN LA COLUMNA 8
```

Tal y como se ve, el hecho de codificar o decodificar el fichero antes de treabajar con él, no afecta en absoluto a los resultados del programa, que sigue siendo capaz de mostrar con eficacia la información que se le pide.

8.) COGER SEMILLA

Se le pide al usuario que introduzca un número de entre 0 y 9999. Probemos si esto funciona bien y que pasa si se introduce algo que no es un número, si se pulsa enter sin escribir nada, o si se escribe un número fuera del rango:

```
-----INTRODUCE UNA SEMILLA DE CODIFICACIÓN (Nº ENTRE 0 Y 9999)-----
-----ESTA SEMILLA GENERARÁ UNA CODIFICACIÓN EN CONCRETO-----
-----

SEMILLA -----> 12345
ESA NO ES UNA OPCIÓN VÁLIDA
SEMILLA -----> 10014
ESA NO ES UNA OPCIÓN VÁLIDA
SEMILLA ----->
ESA NO ES UNA OPCIÓN VÁLIDA
SEMILLA ----->
ESA NO ES UNA OPCIÓN VÁLIDA
SEMILLA ----->
ESA NO ES UNA OPCIÓN VÁLIDA
SEMILLA -----> -1
ESA NO ES UNA OPCIÓN VÁLIDA
SEMILLA -----> 1234

DESEAS HACER ALGO MÁS?

NOMBRE DEL FICHERO CON EL QUE SE ESTÁ TRABAJANDO: Codificado.txt
SEMILLA DE CODIFICACIÓN: 1234
```

Tal y como vemos, hasta que no se introduzca un número contenido en el rango dado por el programa, este no dejará de pedir una semilla al usuario.

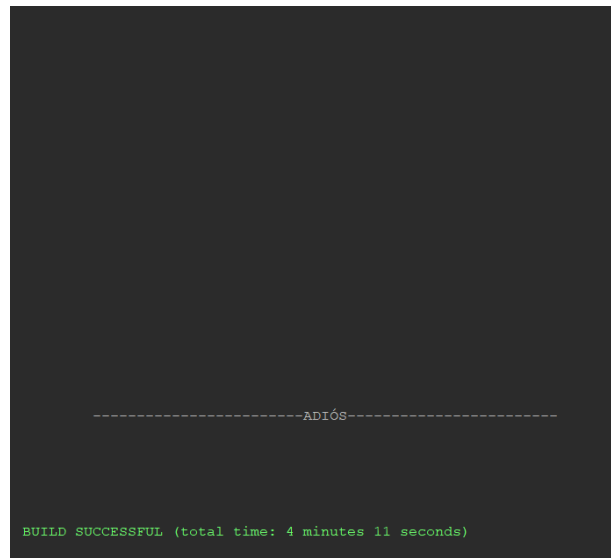
9.) INFORMACIÓN ADICIONAL SOBRE EL FICHERO

Sobre el fichero prueba1.txt, si el usuario selecciona la opción 9 para ver información extra sobre el fichero, el resultado será el siguiente:

```
-----INFORMACIÓN EXTRA SOBRE EL FICHERO-----

1) EL FICHERO CONTIENE UN TOTAL DE 155 LETRAS
2) DE ESAS LETRAS, 74 SON VOCALES Y 81 SON CONSONANTES
3) DE ESAS VOCALES, HAY UN TOTAL DE 0 VOCALES CON ACENTO
4) DE TODAS LAS LETRAS CONTENIDAS EN EL TEXTO, 1 SON MAYÚSCULAS Y 154 SON MINÚSCULAS
5) DE TODOS LOS CARACTERES DEL TEXTO, 0 SON NÚMEROS
6) ADEMÁS, HAY UN TOTAL DE 1 PALABRAS QUE CONTIENEN 5 O MÁS VOCALES
7) LA PALABRA MÁS LARGA DEL TEXTO TIENE UNA LONGITUD DE 10 LETRAS
8) SE HA ENCONTRADO UN TOTAL DE 0 LETRAS CON DIÉRESIS
9) SE HA ENCONTRADO UN TOTAL DE 2 SIGNOS DE PUNTUACIÓN
```

10.) SI EL USUARIO PULSA '0' SE SALE DEL PROGRAMA:



Tal y como vemos, el programa ha sido capaz de llevar a cabo todas las funciones que estaba previsto que hiciera, y se comprueba su eficacia y correcto funcionamiento.

CONCLUSIONES:

Una vez finalizado el programa, hay ciertas cosas que concluimos y que es importante remarcar:

Para empezar, el hecho de haber sido capaz de llevar a cabo 11 tipos diferentes de “ejercicio” o de hacer que las 11 funciones del programa funcionen correctamente sin el uso de variables de tipo String, demuestra que, si se usan las clases correctamente, los tipos base de variable (char, int, float...) son más que suficientes para llevar a cabo una infinidad de problemas o trabajos.

Con el uso de la clase Palabra (o Paraula, el nombre en realidad es indiferente) hemos sido capaces de crear objetos palabra con los que tratar, evitando el uso de Strings y demostrando el potencial que tienen.

Aun así, en algunos casos, habría sido de ayuda poder usarlos ya que en el momento de tratar con objetos palabra y compararlos con el texto, la cantidad de estructuras iterativas, arrays de diferentes tamaños, etc... suponía en cierto modo una dificultad. Por eso, los métodos como el destinado a buscar un texto en el fichero son los que más cuesta de crear.

Además, también hay que destacar la infinidad de problemas que nos permiten resolver unos esquemas de programación tan básicos como el recorrido y la búsqueda, que pese a ser tan simples, nos han ayudado a conseguir en todo momento lo que queríamos y necesitábamos. Pese a que en algunos momentos se vuelve tedioso y puede llegar a liar, si se usan comentarios y se crea el código de forma ordenada, esto no supondrá un problema.

Las estructuras iterativas/reiterativas, o más vulgarmente, “bucles” también han sido vitales para analizar y comparar variables, objetos, o para copiar contenido de arrays en otros de tamaño diferente.

Por último, la importancia del diseño descendente queda presente, ya que, en programas con tantas funcionalidades diferentes, programar todo en bloque generaría un caos absoluto. El hecho de poder repartir todo en diferentes métodos, que cada uno cumpla con su función, y que se invoquen desde esa u otra clase para hacerlos funcionar; hace no solo que se evite código redundante, si no también que se vea todo más limpio, ordenado, y lo mejor de todo: en caso de haber algún error por ejemplo al buscar una palabra en el texto, se sabe que el error está en el método destinado a eso, y por tanto la búsqueda del error se reduce muchísimo al disminuir mucho la cantidad de líneas que pueden contener el error.