

## Estudio de caso: comparación de vocabulario

En este ejercicio, utilizaremos ArrayLists para resolver un problema complejo. Desarrollaremos un programa que leerá dos archivos de texto diferentes y comparará su vocabulario. En particular, determinaremos el conjunto de palabras utilizadas en cada archivo y calcularemos las coincidencias entre ellas. Los investigadores en humanidades a menudo realizan comparaciones de vocabulario en selecciones de texto para responder preguntas como: "¿Bryce Echenique realmente escribió sus obras?", desarrollaremos el programa en etapas:

1. La primera versión leerá los dos archivos e informará las palabras únicas en cada uno. Utilizaremos archivos de prueba cortos para esta etapa.
2. La segunda versión calculará las coincidencias entre los dos archivos (es decir, el conjunto de palabras que aparecen en ambos archivos). Continuaremos usando archivos de prueba cortos para esta etapa.
3. La tercera versión leerá archivos de texto grandes y realizará un análisis de los resultados.

### Algunas consideraciones de eficiencia

Muchos de nuestros programas involucraron cálculos bastante simples y conjuntos de datos bastante pequeños. A medida que empiece a escribir programas más complejos, verá que debe preocuparse si los programas se ejecutan lentamente porque realizan cálculos complejos o manejan grandes cantidades de datos. Necesitamos explorarlo al menos brevemente para este caso de estudio porque de lo contrario es probable que sigamos un enfoque que no funcionará bien. Por ejemplo, en el programa que estamos a punto de escribir, tenemos que leer un archivo y obtener una lista de las palabras del archivo sin duplicados. Un enfoque sería probar cada palabra mientras la leemos para ver si está en la lista, como se describe en el siguiente pseudocódigo:

```
lista = nueva lista vacía.  
while (más palabras para procesar) {  
    palabra = siguiente palabra del archivo  
    if (la lista no contiene palabra) {  
        agregar palabra a la lista  
    }  
}
```

El problema con este enfoque es que requeriría que llamemos al método **contains** de ArrayList cada vez que el programa ejecuta el bucle. Resulta que el método **contains** puede ser bastante costoso para llamar en términos de tiempo. Para averiguar si un valor particular está en la lista, el método tiene que pasar por cada valor diferente en la lista. Entonces, a medida que la lista se hace más y más grande, se vuelve cada vez más costoso buscar a través de ella para ver si contiene una palabra en particular. Nos encontraremos con un problema similar cuando lleguemos a la segunda versión del programa y tengamos que calcular las coincidencias entre las dos listas. La forma más sencilla de calcular las coincidencias sería escribir un método como este:

```
coincidencias = nueva lista vacía  
para (cada palabra en la lista1) {  
    if (la palabra está en la lista2) {  
        agregar palabra a coincidencias  
    }  
}
```

```
}
```

Este enfoque nuevamente requerirá llamar al método **contains** para una lista que podría ser muy grande. Si ambas listas son grandes, entonces el enfoque se ejecutará de manera particularmente lenta. Ambos posibles cuellos de botella pueden abordarse tratando listas ordenadas. En una lista ordenada de palabras, todos los duplicados se agrupan, lo que los hace más fáciles de detectar. Y buscar las coincidencias entre dos listas ordenadas es más fácil que buscar coincidencias en dos listas que no están ordenadas. Por supuesto, la clasificación tampoco es barata. Se necesita una cantidad de tiempo no trivial para ordenar una lista. Pero si podemos ordenar la lista solo una vez, resultará más barato que hacer todas esas llamadas al método **contains**. En lugar de tratar de eliminar los duplicados mientras leemos las palabras, podemos leer todas las palabras directamente en la lista. De esa forma no haremos llamadas costosas con el método **contains**. Después de haber leído todo, podemos poner la lista en orden. Cuando hacemos eso, todos los duplicados aparecerán uno al lado del otro, por lo que podemos deshacernos de ellos con bastante facilidad. Leer todas las palabras en la lista y luego eliminar duplicados requerirá más memoria que eliminar los duplicados a medida que avanzamos, pero terminará ejecutándose más rápido porque la única operación costosa que tendremos es el paso de clasificación. Esta es una compensación clásica entre el tiempo de ejecución y la memoria que aparece a menudo en informática. Podemos hacer que los programas se ejecuten más rápido si estamos dispuestos a usar más memoria o podemos limitar la memoria si no nos importa que el programa tarde más en ejecutarse. Nuestro enfoque para construir la lista de palabras, entonces, será el siguiente:

```
lista = nueva lista vacía
while (hay más palabras para procesar) {
  agregar palabra a la lista}
Ordenar lista
Eliminar duplicados.
```

La tarea de eliminar duplicados también plantea una consideración de eficiencia. Un enfoque obvio sería el siguiente:

```
Para (cada palabra en la lista) {
  if (la palabra es un duplicado) {
    eliminar la palabra de la lista}
}
```

Resulta que eliminar es otra operación costosa. Un mejor enfoque es simplemente crear una nueva lista que no tenga duplicados:

```
resultado = nueva lista vacía
para (cada palabra en la lista) {
  if (la palabra no es un duplicado) {
    agregar palabra a resultado}
}
```

Este código se ejecuta más rápido porque el método que agrega una palabra al final de la lista se ejecuta muy rápido en comparación con el método que elimina una palabra del medio de la

lista. Mientras escribimos el código real, refinaremos el pseudocódigo presentado aquí, pero al menos tenemos una idea del enfoque que vamos a tomar. Para cada archivo, leeremos todas las palabras en una lista y las clasificaremos una vez. Luego, utilizaremos las listas ordenadas para crear listas que no tengan duplicados. Luego usaremos esas dos listas ordenadas para buscar la coincidencia entre las dos listas.

El programa que estamos escribiendo solo será interesante cuando comparemos archivos de entrada grandes, pero mientras estamos desarrollando el programa será más fácil usar archivos de entrada cortos para que podamos verificar fácilmente si estamos obteniendo la respuesta correcta. El uso de archivos de entrada cortos también significa que no tenemos que preocuparnos por el tiempo de ejecución. Cuando usa un archivo de entrada grande y el programa tarda mucho en ejecutarse, es difícil saber si el programa alguna vez terminará de ejecutarse. Si desarrollamos el programa con el uso de archivos de entrada cortos, sabremos que nunca debería llevar mucho tiempo ejecutarlo. Entonces, si accidentalmente introducimos un bucle infinito en nuestro programa, sabremos de inmediato que el problema tiene que ver con nuestro código, no con el hecho de que tenemos muchos datos para procesar.

Usaremos las dos primeras estrofas de una canción popular para niños como nuestros archivos de entrada. Crearemos un archivo llamado test1.txt que contiene el siguiente texto:

El viajar es un placer que nos suele suceder en el auto de papa nos iremos a pasear.

Vamos de paseo, pi pi pi en un auto feo, pi pi pi pero no me importa, pi pi pi porque

llevo torta, pi pi pi.

También usaremos un archivo llamado test2.txt que contiene el siguiente texto:

Atencion, vamos a pasar por un tunel.

Por el tunel pasaras la bocina tocaras la cancion del pi pi pi la cancion del pa pa pa.

Vamos de paseo, pi pi pi en un auto feo, pi pi pi pero no me importa, pi pi pi porque llevo

torta, pi pi pi.

Necesitamos abrir cada uno de estos archivos con un Scanner, por lo que nuestro método principal comenzará con las siguientes líneas de código:

```
Scanner in1 = new Scanner(new File("test1.txt"));
Scanner in2 = new Scanner(new File("test2.txt"));
```

Entonces queremos calcular el vocabulario único contenido en cada archivo. Podemos almacenar esta lista de palabras en un ArrayList <String>. La operación será la misma para cada archivo, por lo que tiene sentido escribir un único método que llamamos dos veces. El método debería tomar el Scanner como parámetro y debería convertirlo en un ArrayList <String> que contiene el vocabulario. Entonces, después de abrir los archivos, podemos ejecutar el siguiente código:

```
ArrayList<String> lista1 = getPalabras(in1);
ArrayList<String> lista2 = getPalabras(in2);
```

Esta versión inicial está destinada a ser bastante simple, por lo que después de calcular el vocabulario para cada archivo, simplemente podemos informarlo:

```
System.out.println("lista1 = " + lista1);
System.out.println("lista2 = " + lista2);
```

El trabajo difícil para esta versión del programa se reduce a escribir el método `getPalabras`. Este método debería leer todas las palabras del `Scanner` (`input`), creando una `ArrayList<String>` que contenga esas palabras y eliminando cualquier duplicado. Recuerde que nuestra primera tarea es leer todas las palabras en una lista y luego ordenar la lista. Para nuestros propósitos, no nos interesan las mayúsculas, por lo que podemos convertir cada palabra a minúsculas antes de agregarla a la lista y luego, debido a que estamos usando una `ArrayList<String>`, podemos llamar a `Collections.sort` para ordenar la lista. Por lo tanto, podemos construir la lista usando el siguiente código:

```
while (input.hasNext()) {
    String next = input.next().toLowerCase();
    palabras.add(next);
}
Collections.sort(palabras);
```

Una vez que la lista ha sido ordenada, los duplicados de cualquier palabra se agruparán. Recuerde que nuestro plan es construir una nueva lista que no tenga duplicados. La forma más sencilla de eliminar duplicados es buscar transiciones entre palabras. Por ejemplo, si tenemos 5 ocurrencias de una palabra seguidas de 10 ocurrencias de otra palabra, la mayoría de los pares de palabras adyacentes serán iguales entre sí. Sin embargo, en el medio de esos pares iguales, cuando hagamos la transición de la primera palabra a la segunda palabra, habrá un par de palabras que no son iguales. Cada vez que vemos esa transición, sabemos que estamos viendo una nueva palabra que debería agregarse a nuestra nueva lista. Buscar transiciones conduce a un problema clásico. Por ejemplo, si hay 10 palabras únicas, habrá 9 transiciones. Podemos resolver el problema agregando la primera palabra antes de que comience el ciclo. Luego podemos buscar palabras que no sean iguales a las palabras que las preceden y agregarlas a la lista. Expresado como pseudocódigo, nuestra solución es la siguiente:

```
Construir una nueva lista vacía
agregue la primera palabra a la nueva lista
para (cada i) {
    if (el valor en i no es igual al valor en i-1) {
        agregar valor en el índice i
    }
}
```

Este esquema se puede convertir en código real de manera bastante directa, pero debemos tener cuidado de comenzar `i` en 1 en lugar de en 0 porque en el bucle comparamos cada palabra con la que viene antes y la primera palabra no tiene nada antes. También debemos tener cuidado de llamar al método `get` de `ArrayList` para obtener valores individuales y usar el método `equals` para comparar cadenas para la igualdad:

```
ArrayList<String> result = new ArrayList<String>();
result.add(palabras.get(0));
for (int i = 1; i < palabras.size(); i++) {
```

```

        if (!palabras.get(i).equals(palabras.get(i - 1))) {
            result.add(palabras.get(i));
        }
    }
}

```

Todavía hay un problema menor con este código: si el archivo de entrada está vacío, no habrá una primera palabra para agregar a la nueva lista. Por lo tanto, necesitamos un `if` extra para asegurarnos de no intentar agregar valores a la nueva lista si el archivo de entrada está vacío. Nuestro programa ahora dice lo siguiente:

```

// Primera versión del programa de vocabulario que lee dos
// archivos y determina las palabras únicas en cada uno
import java.util.*;
import java.io.*;
public class Vocabulario1 {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner in1 = new Scanner(new File("test1.txt"));
        Scanner in2 = new Scanner(new File("test2.txt"));
        ArrayList<String> lista1 = getPalabras(in1);
        ArrayList<String> lista2 = getPalabras(in2);
        System.out.println("lista1 = " + lista1);
        System.out.println("lista2 = " + lista2);
    }
    public static ArrayList<String> getPalabras(Scanner input) {
        // lee todas las palabras y los ordena
        ArrayList<String> palabras = new ArrayList<String>();
        while (input.hasNext()) {
            String next = input.next().toLowerCase();
            palabras.add(next);
        }
        Collections.sort(palabras);
        // agregue palabras únicas a la nueva lista y la devuelve
        ArrayList<String> result = new ArrayList<String>();
        if (palabras.size() > 0) {
            result.add(palabras.get(0));
            for (int i = 1; i < palabras.size(); i++) {
                if (!palabras.get(i).equals(palabras.get(i-1))) {
                    result.add(palabras.get(i));
                }
            }
        }
        return result;
    }
}

```

El programa produce el siguiente resultado:

```

lista1 = [a, auto, de, el, en, es, feo,, importa,, iremos, llevo, me, no, nos, papa, pasear., paseo,, pero, pi, pi., placer,
porque, que, suceder, suele, torta,, un, vamos, viajar]

```

```
lista2 = [a, atencion,, auto, bocina, canción, de, del, el, en, feo,, importa,, la, llevo, me, no, pa, pa., pasar, pasaras, paseo,, pero, pi, pi., por, porque, tocaras, torta,, tunel, tunel., un, vamos]
```

Hemos reducido el primer archivo a 28 palabras únicas y el segundo a 31 palabras únicas. El programa ignora correctamente las diferencias de mayúsculas (Vamos), pero no ignora las diferencias en la puntuación. Por ejemplo, considera "paseo," y "paseo" ser palabras diferentes. Del mismo modo, considera "pi." (con un punto) y "pi" (sin un punto) como palabras diferentes. Solucionaremos ese problema en la Versión 3 de nuestro programa.

## Versión 2: Cálculo de las coincidencias

La primera versión del programa produce dos ArrayLists ordenados que contienen conjuntos de palabras únicas. Para la segunda versión, queremos calcular las coincidencias entre las dos listas de palabras y reportarlo. Esta operación es lo suficientemente compleja como para merecer estar en su propio método. Entonces, agregaremos la siguiente línea de código al método principal justo después de que se construyan las dos listas de palabras:

```
ArrayList <String> comun = getCoincidencias (lista1, lista2);
```

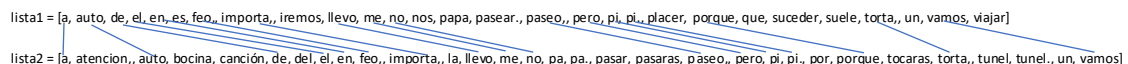
La tarea principal para la segunda versión de nuestro programa es implementar el método `getCoincidencias`. Mire detenidamente las dos listas de palabras producidas por la primera versión:

```
lista1 = [a, auto, de, el, en, es, feo,, importa,, iremos, llevo, me, no, nos, papa, pasear., paseo,, pero, pi, pi., placer, porque, que, suceder, suele, torta,, un, vamos, viajar]
```

```
lista2 = [a, atencion,, auto, bocina, canción, de, del, el, en, feo,, importa,, la, llevo, me, no, pa, pa., pasar, pasaras, paseo,, pero, pi, pi., por, porque, tocaras, torta,, tunel, tunel., un, vamos]
```

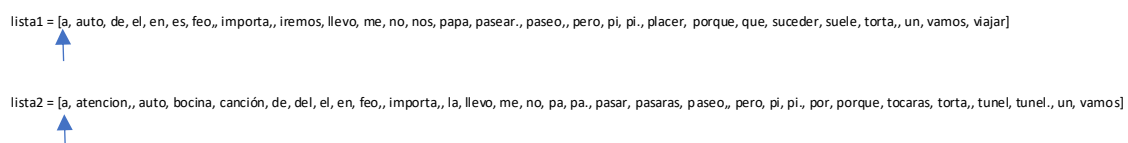
Las personas son bastante buenas para encontrar coincidencias, por lo que probablemente pueda ver exactamente qué palabras coinciden. Ambas listas comienzan con "a", por lo que es parte de las coincidencias. Saltando la palabra "atencion," en la segunda lista, encontramos que la siguiente coincidencia es para la palabra "auto". Luego tenemos tres coincidencias más con las palabras "de" "el" "en". El conjunto completo de coincidencias es el siguiente:

```
lista1 = [a, auto, de, el, en, es, feo,, importa,, iremos, llevo, me, no, nos, papa, pasear., paseo,, pero, pi, pi., placer, porque, que, suceder, suele, torta,, un, vamos, viajar]
lista2 = [a, atencion,, auto, bocina, canción, de, del, el, en, feo,, importa,, la, llevo, me, no, pa, pa., pasar, pasaras, paseo,, pero, pi, pi., por, porque, tocaras, torta,, tunel, tunel., un, vamos]
```



Queremos diseñar un algoritmo que sea paralelo a lo que hacen las personas cuando buscan coincidencias. Imagine poner un dedo de su mano izquierda en la primera lista y poner un dedo de su mano derecha en la segunda lista para realizar un seguimiento de dónde se encuentra en cada lista. Compararemos las palabras a las que está apuntando y, según cómo se comparen, moveremos uno o ambos dedos hacia adelante. Comenzamos con el dedo izquierdo en la palabra "a" en la primera lista y el dedo derecho en la palabra "a" en la segunda lista.

```
lista1 = [a, auto, de, el, en, es, feo,, importa,, iremos, llevo, me, no, nos, papa, pasear., paseo,, pero, pi, pi., placer, porque, que, suceder, suele, torta,, un, vamos, viajar]
lista2 = [a, atencion,, auto, bocina, canción, de, del, el, en, feo,, importa,, la, llevo, me, no, pa, pa., pasar, pasaras, paseo,, pero, pi, pi., por, porque, tocaras, torta,, tunel, tunel., un, vamos]
```



Las palabras coinciden, por lo que agregamos esa palabra a la lista de coincidencias y movemos ambos dedos hacia adelante:

lista1 = [a, auto, de, el, en, es, feo,, importa,, iremos, llevo, me, no, nos, papa, pasear,, paseo,, pero, pi, pi., placer, porque, que, suceder, suele, torta,, un, vamos, viajar]



lista2 = [a, atencion,, auto, bocina, canción, de, del, el, en, feo,, importa,, la, llevo, me, no, pa, pa., pasar, pasaras, paseo,, pero, pi, pi., por, porque, tocaras, torta,, tunel, tunel., un, vamos]



Ahora estamos señalando la palabra "auto" de la primera lista y la palabra "atención," de la segunda lista. Las palabras no coinciden. Entonces, ¿Qué haces? Resulta que la palabra "auto" en lista1 va a coincidir con la misma palabra en lista2. Entonces, ¿cómo sabes si debes mover el dedo izquierdo o derecho hacia adelante? Debido a que las listas están ordenadas y porque la palabra "atencion," viene antes de la palabra "auto", sabemos que no puede haber una coincidencia para la palabra "atención," en la primera lista. Cada palabra que viene después de "auto" en la primera lista será alfabéticamente mayor que "atencion,", por lo que la palabra "atencion," no pueden estar allí. Por lo tanto, podemos mover el dedo derecho hacia adelante para omitir la palabra "atencion,":

lista1 = [a, auto, de, el, en, es, feo,, importa,, iremos, llevo, me, no, nos, papa, pasear,, paseo,, pero, pi, pi., placer, porque, que, suceder, suele, torta,, un, vamos, viajar]



lista2 = [a, atencion,, auto, bocina, canción, de, del, el, en, feo,, importa,, la, llevo, me, no, pa, pa., pasar, pasaras, paseo,, pero, pi, pi., por, porque, tocaras, torta,, tunel, tunel., un, vamos]



Esto nos lleva a la segunda coincidencia, para "auto", y el algoritmo continúa. En general, nos encontramos en una de tres situaciones cuando comparamos la palabra actual en lista1 con la palabra actual en lista2:

- Las palabras pueden ser iguales, en cuyo caso hemos encontrado una coincidencia que debe incluirse en la lista de coincidencias y debemos avanzar a la siguiente palabra en cada lista.
- La palabra de la primera lista puede ser alfabéticamente menor que la palabra de la segunda lista, en cuyo caso podemos omitirla porque no puede coincidir con nada en la segunda lista.
- La palabra de la segunda lista puede ser alfabéticamente menor que la palabra de la primera lista, en cuyo caso podemos omitirla porque no puede coincidir con nada en la primera lista.

Por lo tanto, el enfoque básico que queremos usar se puede describir con el siguiente pseudocódigo:

```
if (la palabra de la lista1 es igual a la palabra de la lista2) {  
    coincidencia de registro.  
    saltar palabra en cada lista.  
} else if (palabra de la lista1 < palabra de la lista2) {  
    omitir palabra en la lista1.  
} else {  
    omitir palabra en la lista2.  
}
```

Podemos refinar este pseudocódigo introduciendo dos variables de índice y colocando este código dentro de un bucle:

i1 = 0.

```

i2 = 0.
while (más valores para comparar) {
    if (list1.get (i1) es igual a list2.get (i2)) {
        registro de coincidencia.
        incremento i1.
        incremento i2.
    } else if (list.get (i1) menor que list.get (i2)) {
        incremento i1.
    } else {
        incremento i2.
    }
}

```

Esta versión del pseudocódigo ahora está bastante cerca del código real. Primero, tenemos que encontrar una prueba de bucle apropiada. Comenzamos las dos variables de índice en 0 e incrementamos una o ambas cada vez a través del ciclo. Eventualmente, se nos agotarán los valores en una o ambas listas, y cuando eso suceda, no habrá más coincidencias para encontrar. Entonces, queremos continuar en el ciclo while siempre que las dos variables de índice no hayan llegado al final de la lista. También tenemos que descubrir cómo comparar las dos palabras. Debido a que la clase String implementa la interfaz Comparable, podemos usar su método **compareTo**. Finalmente, tenemos que construir una ArrayList para almacenar las coincidencias y tenemos que devolver esta lista después de que el ciclo haya completado la ejecución. Por lo tanto, podemos convertir nuestro pseudocódigo en el siguiente código real:

```

ArrayList<String> result = new ArrayList<String>();
int i1 = 0;
int i2 = 0;
while (i1 < list1.size() && i2 < list2.size()) {
    int num = list1.get(i1).compareTo(list2.get(i2));
    if (num == 0) {
        result.add(list1.get(i1));
        i1++;
        i2++;
    } else if (num < 0) {
        i1++;
    } else { // num > 0
        i2++;
    }
}
return result;

```

Después de convertir este código en un método y modificar main para llamar al método e informar las coincidencias, terminamos con la siguiente nueva versión del programa:

```

// Segunda versión del programa de vocabulario que lee
// dos archivos e informa la coincidencias entre ellos

```



```

import java.util.*;
import java.io.*;
public class Vocabulario2 {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner in1 = new Scanner(new File("test1.txt"));
        Scanner in2 = new Scanner(new File("test2.txt"));
        ArrayList<String> lista1 = getPalabras(in1);
        ArrayList<String> lista2 = getPalabras(in2);
        System.out.println("lista1 = " + lista1);
        System.out.println("lista2 = " + lista2);
        ArrayList<String> comun = getCoincidencias(lista1, lista2);
        System.out.println("Coincidencias = " + comun);
    }
    public static ArrayList<String> getPalabras(Scanner input) {
        // lee todas las palabras y los ordena
        ArrayList<String> palabras = new ArrayList<String>();
        while (input.hasNext()) {
            String next = input.next().toLowerCase();
            palabras.add(next);}
        Collections.sort(palabras);
        // agregue palabras únicas a la nueva lista y la devuelve
        ArrayList<String> result = new ArrayList<String>();
        if (palabras.size() > 0) {
            result.add(palabras.get(0));
            for (int i = 1; i < palabras.size(); i++) {
                if (!palabras.get(i).equals(palabras.get(i - 1))) {
                    result.add(palabras.get(i)); }
            }
        }
        return result;
    }
    public static ArrayList<String> getCoincidencias(ArrayList<String> lista1, ArrayList<String>
lista2) {
        ArrayList<String> result = new ArrayList<String>();
        int i1 = 0;
        int i2 = 0;
        while (i1 < lista1.size() && i2 < lista2.size()) {
            int num = lista1.get(i1).compareTo(lista2.get(i2));
            if (num == 0) {
                result.add(lista1.get(i1));
                i1++;
                i2++;
            } else if (num < 0) {
                i1++;
            } else { // num > 0
                i2++;
            }
        }
        return result;
    }
}

```

```
}  
}
```

Esta versión del programa produce el siguiente resultado:

```
lista1 = [a, auto, de, el, en, es, feo,, importa,, iremos, llevo, me, no, nos, papa, pasear., paseo,, pero, pi, pi., placer,  
porque, que, suceder, suele, torta,, un, vamos, viajar]
```

```
lista2 = [a, atencion,, auto, bocina, canción, de, del, el, en, feo,, importa,, la, llevo, me, no, pa, pa., pasar, pasaras,  
paseo,,pero, pi, pi., por, porque, tocaras, torta,, tunel, tunel., un, vamos]
```

```
Coincidencias = [a, auto, de, el, en, feo,, importa,, llevo, me, no, paseo,, pero, pi, pi., porque, torta,, un, vamos]
```

### Versión 3: Programa completo

Nuestro programa ahora crea correctamente una lista de vocabulario para cada uno de los dos archivos y calcula las coincidencias entre ellos. El programa imprimió las tres listas de palabras, pero eso no será muy conveniente para archivos de texto grandes que contienen miles de palabras diferentes. Preferiríamos que el programa informara estadísticas generales, incluyendo el número de palabras en cada lista, el número de palabras coincidentes y el porcentaje de coincidencia. El programa también debe contener al menos una breve introducción para explicar lo que hace, y podemos escribirlo de modo que solicite nombres de archivo en lugar de usar nombres de archivo en el código. Esto también parece un buen momento para pensar en la puntuación. Las dos primeras versiones permitieron que las palabras contengan caracteres de puntuación como comas, puntos y guiones que normalmente no consideraríamos parte de una palabra. Podemos mejorar nuestra solución diciéndole al Scanner qué partes del archivo de entrada debe ignorar. Los objetos Scanner tienen un método llamado `useDelimiter` que puede llamar para decirles qué caracteres usar cuando divide el archivo de entrada en tokens. Cuando llama al método, le pasa lo que se conoce como una expresión regular. Las expresiones regulares son una forma muy flexible de describir patrones de caracteres. Hay algo de documentación sobre ellos en las páginas API para la clase llamada `Pattern`. Para nuestros propósitos, queremos formar una expresión regular que indique al Scanner que mire solo los caracteres que son parte de lo que consideramos palabras. Es decir, queremos que el Scanner capture letras y apóstrofes. La siguiente expresión regular es un buen punto de partida:

```
[a-zA-Z ']
```

Esta expresión regular se leería como, "Cualquier carácter en el rango de a a z, en el rango de A a Z o un apóstrofe". Esta es una buena descripción del tipo de caracteres que queremos que incluya el Scanner. Pero en realidad necesitamos decirle al Scanner qué caracteres ignorar, por lo que debemos indicar que debe usar el conjunto opuesto de caracteres. La manera fácil de hacerlo es incluir el símbolo `^` delante de la lista de caracteres legales:

```
[^a-zA-Z ']
```

Esta expresión regular se leería como: "Cualquier carácter que no sean los caracteres que están en el rango de la a a la z, en el rango de la A a la Z o un apóstrofe". Sin embargo, la expresión no es del todo correcta, ya que puede haber muchos de esos caracteres seguidos. Por ejemplo, puede haber varios espacios, guiones u otros caracteres de puntuación que separan dos palabras. Podemos indicar que una secuencia de caracteres ilegales debe ignorarse poniendo un

signo más después de los corchetes para indicar "Cualquier secuencia de uno o más de estos caracteres":

```
[^a-zA-Z'] +
```

Pasamos esta expresión regular como una cadena a una llamada en useDelimiter. Podemos agregar esto al comienzo del método getPalabras:

```
public static ArrayList<String> getPalabras(Scanner input) {  
    input.useDelimiter("[^a-zA-Z']+");  
    ...  
}
```

El siguiente es un programa completo que incorpora todos estos cambios e incluye comentarios más extensos:

```
// Este programa lee dos archivos de texto y compara el  
// vocabulario utilizado en cada uno.  
import java.util.*;  
import java.io.*;  
  
public class Main {  
    public static void main(String[] args) throws FileNotFoundException {  
        Scanner console = new Scanner(System.in);  
        darIntro();  
  
        System.out.print("Archivo #1 nombre? ");  
        Scanner in1 = new Scanner(new File(console.nextLine()));  
        System.out.print("Archivo #2 nombre? ");  
        Scanner in2 = new Scanner(new File(console.nextLine()));  
        System.out.println();  
        ArrayList<String> lista1 = getPalabras(in1);  
        ArrayList<String> lista2 = getPalabras(in2);  
        ArrayList<String> comun = getCoincidencias(lista1, lista2);  
  
        reportarResultados(lista1, lista2, comun);  
    }  
  
    // post: lee palabras del Scanner, las convierte en minúsculas  
    // devuelve una lista ordenada de palabras únicas  
  
    public static ArrayList<String> getPalabras(Scanner input) {  
        // ignora todos los caracteres excepto los alfabéticos y apóstrofes  
        input.useDelimiter("[^a-zA-Z']+");  
  
        // lee todas las palabras y los ordena  
        ArrayList<String> palabras = new ArrayList<String>();  
        while (input.hasNext()) {
```

```

        String next = input.next().toLowerCase();
        palabras.add(next);
    }
    Collections.sort(palabras);
    // agregue palabras únicas a la nueva lista y la devuelve
    ArrayList<String> result = new ArrayList<String>();
    if (palabras.size() > 0) {
        result.add(palabras.get(0));
        for (int i = 1; i < palabras.size(); i++) {
            if (!palabras.get(i).equals(palabras.get(i - 1))) {
                result.add(palabras.get(i));
            }
        }
    }
    return result;
}

// pre: list1 y list2 están ordenados y no tienen duplicados
// post: construye y devuelve una ArrayList que contiene
// las palabras en común entre list1 y list2

public static ArrayList<String> getCoincidencias(ArrayList<String>
lista1, ArrayList<String> lista2) {
    ArrayList<String> result = new ArrayList<String>();
    int i1 = 0;
    int i2 = 0;
    while (i1 < lista1.size() && i2 < lista2.size()) {
        int num = lista1.get(i1).compareTo(lista2.get(i2));
        if (num == 0) {
            result.add(lista1.get(i1));
            i1++;
            i2++;
        } else if (num < 0) {
            i1++;
        } else { // num > 0
            i2++;
        }
    }
    return result;
}

// post: explica el programa al usuario
public static void darIntro() {
    System.out.println("Este programa compara dos archivos de texto");
    System.out.println("e informa el número de palabras en");
    System.out.println("común y el porcentaje de coincidencias");
    System.out.println();
}
// pre: comun contiene las coincidencias entre lista1 y lista2

```

```
// post: informa estadísticas sobre las listas y sus coincidencias

    public static void reportarResultados(ArrayList<String> lista1,
ArrayList<String> lista2, ArrayList<String> comun) {
        System.out.println("Archivo #1 palabras = " + lista1.size());
        System.out.println("Archivo #2 palabras = " + lista2.size());
        System.out.println("Palabras en común = " + comun.size());
        double pct1 = 100.0 * comun.size() / lista1.size();
        double pct2 = 100.0 * comun.size() / lista2.size();
        System.out.println("% del archivo 1 en superposición = " + pct1);
        System.out.println("% del archivo 2 en superposición = " + pct2);
    }
}
```

El siguiente resultado es una ejecución del programa que compara los archivos descritos en las versiones anteriores:

Este programa compara dos archivos de texto

e informa el número de palabras en

común y el porcentaje de coincidencias

Archivo #1 nombre? test1.txt

Archivo #2 nombre? test2.txt

Archivo #1 palabras = 27

Archivo #2 palabras = 29

Palabras en común = 17

% del archivo 1 en superposición = 62.96296296296296

% del archivo 2 en superposición = 58.62068965517241

### Ejercicios:

1. Escriba una versión modificada del programa de vocabulario desarrollado que utilice Sets en lugar de ArrayLists para almacenar sus palabras. (¡El programa será notablemente más corto y se ejecutará más rápido!).
2. Probar el programa con un par de obras literarias de un mismo autor y considerar los caracteres con tildes.
3. Desarrollar el estudio de caso: Votación por orden de preferencia, y el proyecto de programación número 5 del capítulo 10 del libro Building Java Programs.