



Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación
CC3001 – Algoritmos y Estructuras de Datos

TAREA 2

Diff para archivos

Nombre Alumno : Gabriel Iturra Bocaz

Correo : gabrieliturra@ug.uchile.cl

Profesor : Patricio Poblete

Profesor Auxiliar : Manuel Cáceres
Sergio Peñafiel

Fecha : 17 de Abril de 2016
Santiago, Chile.

Contenido

I.	Introducción	1
II.	Análisis del Problema	2
II.1.	Distancia de Levenshtein.....	2
II.2.	Problema Propuesto	3
III.	Solución del Problema.....	4
III.1.	Desarrollo	4
IV.	Ejemplos de Funcionamiento	6
V.	Resultados y Análisis.....	7

I. Introducción

El siguiente informe detalla el desarrollo de la Tarea 2, del curso Algoritmos y Estructuras de Datos (CC3001). Como segunda tarea del curso, se pretende que el alumno se interiorice con los conceptos de Programación Dinámica y Tabulación de datos. La Programación Dinámica sigue el paradigma “Dividir para reinar”, separando un gran problema en muchos subproblemas, cuyas soluciones se van tabulando y son utilizadas para resolver en conjunto el problema.

La Tarea 2, plantea la implementación de un comando Diff (similar al de Unix) para archivos, el cual se deberá calcular el número de modificación a realizar a un archivo.txt A para convertirlo en un archivo.txt B, además de indicar cuales serán dichas modificaciones en cada una de sus líneas, siendo éstas: Eliminación, Inserción y Sustitución de líneas.

Como una solución utilizando la fuerza bruta es bastante lenta, se recurrió a una implementación de la Distancia de Levenshtein (o Distancia de edición de Strings) que siguiendo un enfoque de Programación Dinámica se encontró una solución más eficiente al problema pedido. Además, de usar los comandos y clases necesarias para apertura, lectura (File, FileReader, ArrayList, etc.) o separación de líneas que tiene JAVA.

.

II. Análisis del Problema

II.1. Distancia de Levenshtein

El problema propuesto tiene como objetivo la implementación de un comando Diff para archivos que y detalle las operaciones a realizar para convertir un archivo A.txt a uno B.txt, y que además, calcule el número de éstas. Para ello, se recurrió a la Distancia de Levenshtein, o también llamada Distancia de Edición de Strings. Esta función matemática calcular recursivamente el número de ediciones que se debe hacer a un string a para convertirlo en un string b. Esta función toma dos string a y b de largos m y n respectivamente, generando una matriz de m+1 x n+1, donde su último elemento corresponde al número de ediciones. En la figura 1, podemos apreciar la forma matemática de la Distancia de Levenshtein.

$$\text{lev}_{a,b}(i, j) = \begin{cases} \text{máx}(i, j) & \text{si } \text{mín}(i, j) = 0 \\ \text{mín} \begin{cases} \text{lev}_{a,b}(i-1, j) + 1 \\ \text{lev}_{a,b}(i, j-1) + 1 \\ \text{lev}_{a,b}(i-1, j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{si no} \end{cases}$$

Figura 1

Es importante mencionar que el primer elemento recursivo dentro de la función mínimo corresponde a realizar un borrado de a a b, el segundo mínimo, cuando se necesita realizar una inserción de a a b, y el último indica cuando se debe hacer una sustitución. En las figuras 2 y 3, podemos apreciar un ejemplo de la distancia entre los string mojados y moteado, y una matriz generada por la función comparando los strings difícil y fácil.

Por ejemplo, la distancia entre **mojados** y **moteado** es 3:

1. **mojados** → **mojado_** (eliminar s)
2. **mojado** → **motado** (sustituir **j** por **t**)
3. **motado** → **moteado** (insertar **e**)

Figura 2

		F	A	C	I	L
	0	1	2	3	4	5
D	1	1	2	3	4	5
I	2	2	2	3	3	4
F	3	2	3	3	4	4
I	4	3	3	4	3	4
C	5	4	4	3	4	4
I	6	5	5	4	3	4
L	7	6	6	5	4	3

Figura 3: Matriz de Levenshtein

II.2. Problema Propuesto

Para su implementación, se debió modificar la Distancia de Levenshtein haciéndola funcional para archivos, encontrando que líneas que se deben modificar, eliminar o insertar siguiendo el cálculo recursivo. En las Figuras 4 y 5, podemos ver un ejemplo del objetivo del problema propuesto.

Pollo	Pato
Perro	Perro
	Conejo
A.txt	B.txt

Figura 4: Archivos de Ejemplo.

```
>java Diff A.txt B.txt
>1. Pollo      -> Pato
   3.          -> Conejo
Distance = 2
```

Figura 5: Output esperado.

III. Solución del Problema

III.1. Desarrollo

Para encontrar la solución al problema planteado, fueron necesario cuatro elementos claves que serán explicados a continuación y describen la implementación del archivo Diff.java.

- Apertura y lectura de archivos: En este punto fue necesario llamar y utilizar los métodos de las clases predefinidas de JAVA, File, FileReader y BufferedReader. Para dar con posibles excepciones y errores que pudiesen generar la apertura y lectura de archivos se usó las clases IOException y FileNotFoundException que combinadas en un bloque try-catch permitieron una buena lectura de archivos, evitando que el programa se cayera.
- Distancia de Levenshtein para archivos: para generar la matriz de la función de Levenshtein, fue necesario ir guardando cada línea de un archivo en un objeto ArrayList (Estos objetos guardan elementos de forma similar a un arreglo, pero tienen la ventaja de poseer un método llamado `arraylist.add(elemento)` que permite ir guardando elementos al final de la lista, pudiendo ser llamados con el método `.get(índice)` a través de un índice.). Luego utilizando 2 bucles for, partiendo de un contador i, que denotaba la cantidad de filas, y un contador j para las columnas, pudiese ser llenadas la primera fila y columna de la matriz de Levenshtein. Finalmente, utilizando otros dos bucles for consecutivos se iba calculando el resto de la matriz siguiendo la formula recursiva de mínimos.

En la figura 6 y 7, se puede ver como se agregaba cada línea a su ArrayList respectivo y la programación de la función de Levenshtein.

```
String linea;  
while ((linea=b1.readLine()) != null) {  
    array1.add(linea);  
  
while ((linea=b2.readLine()) != null) {  
    array2.add(linea);  
}
```

Figura 6: Separación de líneas.

```

int[][] matriz = new int[array1.size()+1][array2.size()+1];
for (int i = 0; i<=array1.size();++i){ // Implementación de la Distancia de Levenshtein
    matriz[i][0] = i;}
for (int j = 0; j<=array2.size();++j){
    matriz[0][j] = j;}
for (int i = 1; i<=array1.size();++i){
    for (int j = 1; j<=array2.size();++j){

        matriz[i][j] = Min(matriz[i-1][j]+1,matriz[i][j-1]+1,matriz[i-1][j-1]+((array1.get(i-1).equals(array2.get(j-1)))?0:1));
    }
}

```

Figura 7: implementación Distancia de Levenshtein.

- Recorrido de la matriz: para generar el output pedido, fue necesario a través de un ciclo while, recorrer la matriz desde su último elemento siguiendo su camino mínimo e ir guardando dentro de un ArrayList, la operación respectiva que se debe hacer dependiendo de las modificaciones para transformar el archivo A al archivo B (estas se identifican siguiendo el cálculo recursivo dentro de lo mínimos en la Función de Levenshtein). Para seguir el camino mínimo se comparaba de un índice (i,j) con el mínimo de los índices (i-1,j-1), (i-1,j) y (i,j-1), si estos eran iguales se recorre la matriz diagonalmente, bajando el valor de los índices, si no fueran iguales se entra se entra en la comparación de los mínimos de la función de Levenshtein, para así identificar el tipo de operación. En la figura 8, se ve puede ver el código.

```

int i = array1.size(), j = array2.size();
while (i>0 && j>0){ // Este while sirve para recorrer el camino mínimo de la matriz,
    // desde el último elemento de la matriz ubicado en la esquina inferior derecha

    if (matriz[i][j]==Min(matriz[i-1][j],matriz[i][j-1],matriz[i-1][j-1])){
        if (matriz[i-1][j-1]!=matriz[i-1][j] && matriz[i-1][j-1]!=matriz[i][j-1]){
            i--;
            j--;}
        else if (matriz[i-1][j-1]==matriz[i][j-1]){
            j--;}
        else if (matriz[i-1][j-1]==matriz[i-1][j]){
            i--;}
    }

    else if (matriz[i][j]!=Min(matriz[i-1][j],matriz[i][j-1],matriz[i-1][j-1])){
        if (matriz[i][j]==matriz[i-1][j]+1){
            array3.add(array1.indexOf(array1.get(i-1))+1+" "+array1.get(i-1)+" -> "+" ");
            i--;
        }
        else if (matriz[i][j]==matriz[i][j-1]+1){
            array3.add(array2.indexOf(array2.get(j-1))+1+" "+array2.get(j-1)+" -> "+" ");
            j--;
        }
        else if (matriz[i][j]==matriz[i-1][j-1]+1){
            array3.add(array2.indexOf(array2.get(j-1))+1+" "+array1.get(i-1)+" -> "+array2.get(j-1));
            i--;
            j--;
        }
    }
}
}

```

Figura 8: Recorriendo la Matriz.

- Impresión Output pedido: para imprimir el output como las operaciones fueron guardados dentro un ArrayList, estos se imprimen iterando el objeto con un ciclo for.

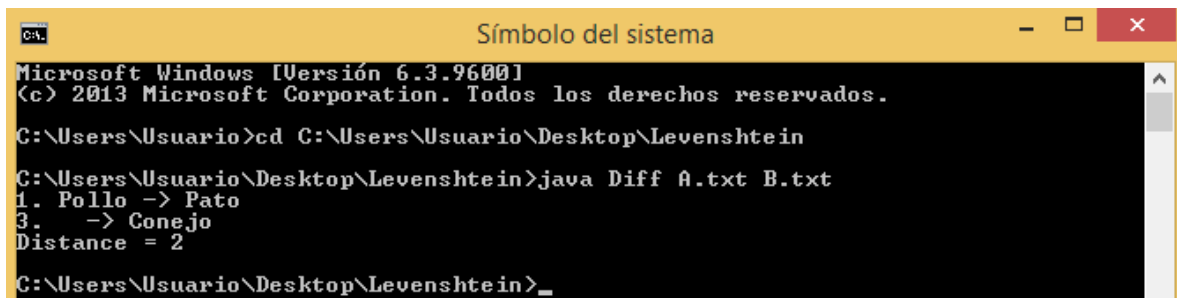
```
for (int k = 0; k<=array3.size()-1;++k){// Se recorre el arreglo para impr
    System.out.print(array3.get((array3.size()-1)-k)+"\n");}
System.out.println("Distance = "+matriz[array1.size()][array2.size()]);}
```

Figura 9

IV. Ejemplos de Funcionamiento

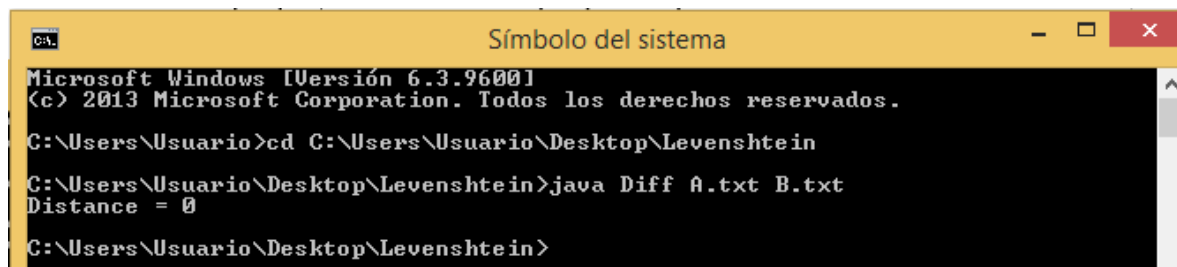
En la siguiente sección de muestra ejemplos de funcionamiento del programa implementado:

- Utilizando los archivos de ejemplo, se muestra el output pedido por el enunciado, donde se identifica una sustitución de Pollo por y una inserción en la tercera línea.



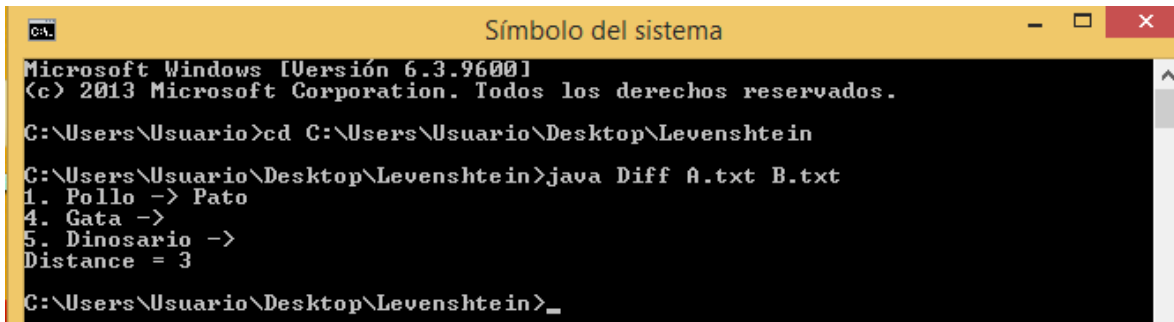
```
C:\Users\Usuario>cd C:\Users\Usuario\Desktop\Levenshtein
C:\Users\Usuario\Desktop\Levenshtein>java Diff A.txt B.txt
1. Pollo -> Pato
3.  -> Conejo
Distance = 2
C:\Users\Usuario\Desktop\Levenshtein>_
```

- Comparación de dos archivos exactamente, iguales donde la distancia es cero y no se identifican operaciones.



```
C:\Users\Usuario>cd C:\Users\Usuario\Desktop\Levenshtein
C:\Users\Usuario\Desktop\Levenshtein>java Diff A.txt B.txt
Distance = 0
C:\Users\Usuario\Desktop\Levenshtein>
```


- Este caso se puede apreciar que se debe eliminar las líneas 4 y 5, para que el archivo A.txt sea igual a B.txt.



```
Microsoft Windows [Versión 6.3.9600]
(c) 2013 Microsoft Corporation. Todos los derechos reservados.

C:\Users\Usuario>cd C:\Users\Usuario\Desktop\Levenshtein
C:\Users\Usuario\Desktop\Levenshtein>java Diff A.txt B.txt
1. Pollo -> Pato
4. Gata ->
5. Dinosaurio ->
Distance = 3
C:\Users\Usuario\Desktop\Levenshtein>_
```

V. Resultados y Conclusiones

Analizando la complejidad computacional del algoritmo, si se hubiese utilizado “fuerza bruta” se habría comparado cada línea del archivo A.txt con cada línea del archivo B.txt, lo que hubiéramos tenido un algoritmo de complejidad $O(m \cdot n)$ pensando que el archivo A.txt tiene m líneas y el archivo B.txt tiene n lo cual es demasiado costoso. Sin embargo, al utilizar La distancia de Levenshtein (en esta caso pasa líneas de archivos.txt), dicha comparación baja a una complejidad de $O(m \cdot n)$ realizando solo una comparación entre las líneas (o informalmente de una pasada), de lo que se puede desprender que es un algoritmos mucho más eficiente que la fuerza bruta y más rápido. Si bien, es mucho más rápido que un algoritmo utilizado por fuerza bruta pueden existir soluciones muchos más óptimos y eficientes de orden lineal para el mismo problema, teniendo una complejidad computacional $O(n)$.

En cuanto a los resultados buscados, se consiguió se realizar el output pedidos utilizando la Distancia de Levenshtein basándose en el cálculo recursivo de sus mínimos, y guiándose por la matriz generada, buscando su camino mínimo siguiendo de acuerdo al número mínimo más cercano en la posición (i,j) .

La Programación Dinámica busca resolver problemas de optimización (ya sea maximización o minimización de alguna función objetivo) logra ser una técnica bastante eficiente para resolver este problema de edición de string, evitando la fuerza bruta, y además de guardar los datos calculados que ser útiles más adelante, como en nuestro caso para identificar el tipo de operación (Sustitución, Inserción o Eliminación).

