Ejercicio 12: Resolución Dinámica

1. d([ ],[ ]) = 0 ya que, al ser ambas cadenas vacías podemos decir que son iguales, por lo tanto, no debe aplicarse ninguna operación para igualarlas.

d(s1,[ ]) = longitud(s1) ya que deben aplicarse tantos borrados como caracteres tenga la cadena s1 para que sea vacía.

d([ ],s2) = longitud(s2) ya que deben aplicarse tantas inserciones en una cadena vacía como caracteres tenga la cadena s2 para lograr igualarlas.

d(s1:c1,s2:c2) = min(d(s1:c1,s2),d(s1,s2:c2),d(s1,s2))+1 buscará de forma recursiva la mínima distancia de edición posible, eligiendo la operación que más convenga en cada caso (borrar, insertar, seleccionar):

INSERCION: d(s1:c1,s2), que “elimina” el elemento c2 de la cadena s2. Este caso se aplicará cuando la cadena s2:c2 tenga mayor longitud que la cadena s1:c1; ¿? ¿? ¿?

BORRADO: d(s1,s2:c2), que elimina el elemento c1 de la cadena s1, reduciendo el problema a transformar s1 en s2:c2. Este caso se aplicará cuando la cadena s1:c1 tenga mayor longitud que s2:c2;

SUSTITUCION: d(s1,s2), que sustituye c1 por c2, reduciendo el problema a transformar s1 en s2, ya que c1 y c2 ya fueron igualados. Este caso se aplica cuando s1:c1 y s2:c2 tienen la misma longitud pero c1 y c2 son distintos. También se aplica cuando c1 y c2 son iguales, pero en este caso el costo no se incrementa ya que no aplica ninguna de las 3 operaciones (borrar, insertar, sustituir).

(Utiliza algoritmo levenshtein)

*- calcular la distancia levenshtein entre dos cadenas*

levenshtein **::** [ **Char** ] **->**  [ **Char** ] **->**  **Int**

levenshtein "" "" **=**  0

levenshtein "" s2 **=**  **longitud** s2

levenshtein s1 "" **=**  **longitud** s1

levenshtein s1 s2

**El |**  **último** s1 **==**  **último** s2 **=** levenshtein ( **init** s1 ) ( **init** s2)

**|**  **de lo contrario**  **=**  **mínimo**  [ 1 **+** levenshtein ( **init** s1 ) s2,

1 **+** levenshtein s1 ( **init** s2 ) ,

1 **+** levenshtein ( **init** s1 ) ( **init** s2 ) ]

El algoritmo que utiliza el ejercicio para resolver este problema es un algoritmo de memorización llamado Top-Down, que inicia al final de ambas cadenas, y finaliza en el inicio de ambas. Para mejorar el tiempo de orden exponencial que este algoritmo genera, primero propondremos un nuevo algoritmo Bottom-Up que inicia al principio de ambas cadenas y las recorre hasta llegar al final: (Utiliza algoritmo del CAP5: 5.7)

CONST MAXCARACTERES = ...;

TYPE CADENA=ARRAY[1..MAXCARACTERES] OF CHAR;

TABLA=ARRAY[0..MAXCARACTERES],[0..MAXCARACTERES] OF CARDINAL;

PROCEDURE Cadena(VAR OB:TABLA;u,v:CADENA;n,m:CARDINAL):CARDINAL;

VAR i,j: CARDINAL;

BEGIN

FOR i:=0 TO m DO OB[i,0]:=i; END;

FOR j:=0 TO n DO OB[0,j]:=j END;

FOR i:=1 TO m DO

FOR j:=1 TO n DO

IF u[i]= v[j] THEN OB[i,j]:=OB[i-1,j-1]

ELSE OB[i,j]:=Min3(OB[i,j-1],OB[i-1,j],OB[i-1,j-1])+1;

END

END

END;

RETURN OB[m,n]

END Cadena;

El problema es que esto es muy lento cuando se compara una cadena larga con una corta.

Finalmente, agregaremos al principio de la implementación anterior la optimización de velocidad para que mejor el tiempo de orden exponencial O(mn). En resumen, compararemos la longitud de ambas cadenas y reduciremos la longitud de la más larga a la misma longitud que la más corta, incrementando el costo en la diferencia de longitud entre ambas (la cantidad de caracteres que se eliminaron), reduciendo el problema a trabajar con 2 cadenas de igual longitud. La implementación finalmente quedaría de la siguiente manera:

CONST MAXCARACTERES = ...;

TYPE CADENA=ARRAY[1..MAXCARACTERES] OF CHAR;

TABLA=ARRAY[0..MAXCARACTERES],[0..MAXCARACTERES] OF CARDINAL;

PROCEDURE Cadena(VAR OB:TABLA;u,v:CADENA;n,m:CARDINAL):CARDINAL;

VAR i,j,d,a: CARDINAL;

OPTIMIZAVEL

| m > n = a = n && m = n && n = a

| m < n = d = n-m && n = m

BEGIN

FOR i:=0 TO m DO OB[i,0]:=i; END;

FOR j:=0 TO n DO OB[0,j]:=j END;

FOR i:=1 TO m DO

FOR j:=1 TO n DO

IF u[i]= v[j] THEN OB[i,j]:=OB[i-1,j-1]

ELSE OB[i,j]:=Min3(OB[i,j-1],OB[i-1,j],OB[i-1,j-1])+1;

END

END

END;

RETURN OB[m,n]

END Cadena;

levenshtein s1 s2 **| longitud** s1 **> longitud** s2 **=** levenshtein s2 s1

**| longitud** s1 **< longitud** s2 **= let** d **= longitud** s2 **- longitud** s1

**en** d **+** levenshtein s1 ( **take** (

**longitud** s2 **-** d ) s2 )

*- la carne del algoritmo*

levenshtein "" "" **=**  0

levenshtein s1 s2 **| último** s1 **== último** s2 **=** levenshtein ( **init** s1 ) ( **init** s2 ) **| de lo contrario = mínimo** [ 1 **+** levenshtein ( **init** s1 ) s2,

1 **+** levenshtein s1 ( **init**

s2 ) ,

1 **+** levenshtein ( **init** s1 ) ( **init** s2 ) ]

1. Este algoritmo, como ya dijimos, guarda todos los datos en memoria, incluso los que ya no hacen falta porque no volverá a utilizarse. Por esta razón, es posible mejorar su orden de complejidad espacial:

Como hemos visto, los datos que realmente hacen falta para calcular un nuevo costo son solamente estos 3 valores: OB [i, j-1], OB [i-1, j], y OB [i-1, j-1]. Estos valores pueden ser guardados en las variables x, y, z respectivamente y ser modificados a medida que el algoritmo vaya avanzando sobre las cadenas. De este modo, cuando el algoritmo llegue a su fin, no quedará guardada toda la tabla de valores en la memoria, sino que solo quedará registro de las variables x, y, z, con los últimos valores que adquirieron.