Ejercicio 12: Dinámica

**Descripción del problema:**

Las palabras pelota y paleta son muy similares. En efecto, una puede transformarse en otra cambiando solo dos letras. La palabra palta también es similar a paleta porque puede transformarse palta en paleta insertando la letra “e” entre la “l” y la “t”, o equivalentemente puede transformarse paleta en palta eliminando la “e” entre la “l” y la “t”. La distancia de edición de dos palabras s1 y s2, d(s1,s2), se define como el mínimo número de operaciones requeridas para transformar s1 en s2, donde una operación puede ser:

* Sustituir una letra por otra
* Borrar una letra
* Insertar una letra

1. Completar las siguientes relaciones de recurrencia, que son suficientes para definir la distancia de edición:

d([ ],[ ]) = ...

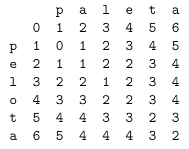
d(s1,[ ]) = ...

d([ ],s2) = ...

d(s1:c1,s2:c2) = min(...)

donde [ ] es la cadena vacía, s1, s2 son cadenas cualesquiera y s:c es la cadena obtenida a partir de s agregando el carácter c al final.

1. Las relaciones de recurrencia del punto anterior indican un procedimiento trivial para calcular la distancia de edición de dos palabras en un tiempo de orden exponencial. Utilice la técnica de programación dinámica para construir un algoritmo que mejore este orden.



Este es un ejemplo de cómo pueden tabularse los resultados de los sub problemas para calcular la distancia de edición entre pelota y paleta.

1. Sugiera cómo puede mejorarse el orden de complejidad espacial de su algoritmo (no lo implemente, es suficiente con una justificación informal).

**Resolución:**

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). Podemos decir que insertar un elemento a la cadena s1:c1 con el objetivo de llegar a igualarla con s2:c2, conlleva el mismo costo que eliminar un elemento de s2:c2 con el objetivo de igualarla con s1:c1. Este último proceso es el que lleva a cabo la operación “inserción”. Esta operación se aplicará cuando la cadena s2:c2 tenga mayor longitud que la cadena s1:c1;

BORRADO: d(s1,s2:c2). 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). 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).

El algoritmo utiliza la técnica de memorización llamada Top-Down, que hace referencia a que recorre las cadenas del final hacia el inicio, y es el siguiente:

d :: [ Char ] -> [ Char ] -> Int

d [] [] = 0

d [] s2 = length s2

d s1 [] = length s1

d s1 s2

| last s1 == last s1 = d (init s1) (init s2)

| otherwise = mínimum [ 1 + d (init s1) s2,

1 + d s1 (init s2),

1. + d (init s1) (init s2) ]

Para mejorar el tiempo de orden exponencial que el algoritmo anterior genera, primero propondremos un nuevo algoritmo Bottom-Up que inicia al principio de ambas cadenas y las recorre hasta llegar al final:

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, por ende, ahora agregaremos, al principio de la implementación anterior, la optimización de velocidad para mejorar 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. Esto nos ahorrará mucho tiempo en el caso de que tengamos dos cadenas con mucha diferencia de longitud. Pero todavía podemos reducir aún más el tiempo del algoritmo.

Hasta ahora, nuestro algoritmo funciona de la siguiente manera: supongamos dos cadenas s1:”pelota” y s2:”paleta”. Uno de nuestros contadores se para en la “p” de s1 mientras que otro contador recorre cada caracter de s2, y repite la misma secuencia para cada carácter de s1. Pero ahora planteemos un mejor algoritmo de la siguiente manera: tenemos nuestras 2 cadenas s1 y s2. Nos pararemos sobre la “p” de s1 y la compararemos con la “p” de s2: no se le aplicará ninguna operación porque son iguales: el costo no se incrementa y el algoritmo continuará comparando las siguientes letras de ambas cadenas, es decir, la “e” y la “a”: se deberá sustituir la “e” por la “a”, incrementar el costo en uno, y continuar de la misma forma con el resto de las cadenas. Nótese la importancia de que ambas cadenas tengan la misma longitud. De esta forma evitamos recorrer tantas veces la segunda cadena, ahorrando mucho tiempo. La implementación de este nuevo algoritmo 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

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

ELSE OB[i,i]:=OB[i,i]+1

END

j:=j+1

END;

RETURN OB[m,m]

END Cadena;

Este algoritmo, guarda todos los datos en memoria, incluso los que ya no hacen falta porque no volverán 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.