

Técnicas Fundamentales

Programación Dinámica II

Segundo semestre 2022

IIC2283

Prof. Nicolás Van Sint Jan

Programación dinámica: un primer ingrediente

Al igual que dividir para conquistar, la técnica de **programación dinámica** resuelve un problema dividiéndolo en sub-problemas más pequeños.

Pero a diferencia de dividir para conquistar, en este caso se espera que **los sub-problemas estén traslapados**.

De esta forma se reduce el número de sub-problemas a resolver, de hecho se espera que este número sea pequeño (al menos polinomial).

Contando el número de caminos en un grafo

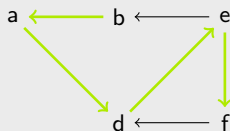
Sea $G = (V, E)$ un **grafo dirigido**.

Recordar que una secuencia v_1, \dots, v_ℓ de elementos en N es un **camino** en G si:

1. $\ell \geq 2$
2. $(v_i, v_{i+1}) \in E$ para cada $i \in \{1, \dots, \ell - 1\}$

Decimos que un camino v_1, \dots, v_ℓ va desde v_1 a v_ℓ , y definimos su largo como $(\ell - 1)$, vale decir, el número de aristas en el camino.

Ejemplo



b, a, d, e, f

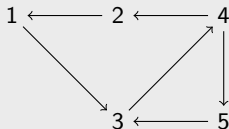
Contando el número de caminos en un grafo

Dado un grafo $G = (V, E)$, un par de nodos v_i, v_f en V y un número ℓ , queremos desarrollar un algoritmo que cuente el **número de caminos** desde v_i a v_f en G cuyo largo es igual a ℓ

Suponemos que $V = \{1, \dots, n\}$, $1 \leq \ell \leq n$ y representamos G a través de su **matriz de adyacencia** M tal que:

Si $(i, j) \in E$, entonces $M[i, j] = 1$, en caso contrario $M[i, j] = 0$.

Ejemplo



$$M = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Una primera definición de **ContarCaminos**

Queremos entonces definir la función **ContarCaminos**(M, v_i, v_f, ℓ).

ContarCaminos($M[1 \dots n][1 \dots n], v_i, v_f, \ell$)

if $\ell = 1$ **then return** $M[v_i, v_f]$

else

$aux := 0$

for $v_j := 1$ **to** n **do**

$aux += M[v_i, v_j] \cdot \text{ContarCaminos}(M, v_j, v_f, \ell - 1)$

return aux

Observe que usamos la notación $C[1 \dots m][1 \dots n]$ para indicar que la matriz C tiene m filas y n columnas.

¿Se puede mejorar este algoritmo?

Una segunda definición de **ContarCaminos**

Podemos reducir el número de llamadas recursivas:

```
ContarCaminos( $M[1 \dots n][1 \dots n]$ ,  $v_i$ ,  $v_f$ ,  $\ell$ )  
  if  $\ell = 1$  then return  $M[v_i, v_f]$   
  else  
     $aux := 0$   
    for  $v_j := 1$  to  $n$  do  
      if  $M[v_i, v_j] = 1$  then  
         $aux += \text{ContarCaminos}(M, v_j, v_f, \ell - 1)$   
  return  $aux$ 
```

¿Hay algún **problema** con este algoritmo?

Outline

Programación dinámica: Grafos

Programación dinámica: Palabras

Outline

Programación dinámica: Grafos

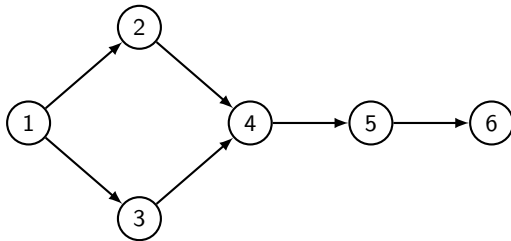
Programación dinámica: Palabras

Llamadas repetidas en **ContarCaminos**

Considere el siguiente grafo G (representado por una matriz M):

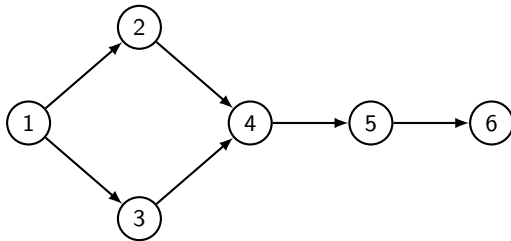
Llamadas repetidas en **ContarCaminos**

Considere el siguiente grafo G (representado por una matriz M):



Llamadas repetidas en **ContarCaminos**

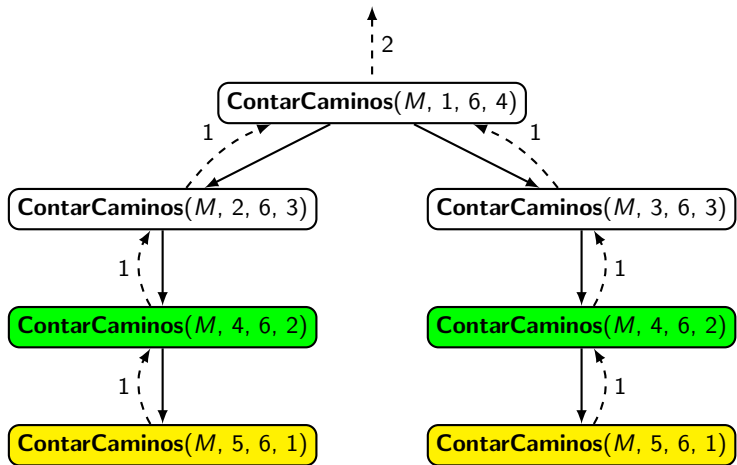
Considere el siguiente grafo G (representado por una matriz M):



Suponga que queremos contar el número de caminos en G desde el nodo 1 al nodo 6 y cuyo largo sea 4

Llamadas repetidas en **ContarCaminos**

Tenemos las siguientes llamadas recursivas:



Llamadas repetidas en **ContarCaminos**

Ejercicio

Demuestre que en el peor caso se deben realizar $\frac{n^\ell - 1}{n - 1}$ llamadas al procedimiento **ContarCaminos**, suponiendo que la entrada es $M[1 \dots n][1 \dots n]$, v_i, v_f, ℓ con $n \geq 2$.

Llamadas repetidas en **ContarCaminos**

Ejercicio

Demuestre que en el peor caso se deben realizar $\frac{n^\ell - 1}{n - 1}$ llamadas al procedimiento **ContarCaminos**, suponiendo que la entrada es $M[1 \dots n][1 \dots n]$, v_i , v_f , ℓ con $n \geq 2$.

- ¿Para qué grafos obtenemos el peor caso?

Llamadas repetidas en **ContarCaminos**

Ejercicio

Demuestre que en el peor caso se deben realizar $\frac{n^\ell - 1}{n - 1}$ llamadas al procedimiento **ContarCaminos**, suponiendo que la entrada es $M[1 \dots n][1 \dots n]$, v_i , v_f , ℓ con $n \geq 2$.

- ¿Para qué grafos obtenemos el peor caso?

El algoritmo realiza **un número exponencial de llamadas repetidas**. Dado que sólo podemos tener $n^2 \cdot \ell$ llamadas distintas en la ejecución de **ContarCaminos**($M[1 \dots n][1 \dots n]$, v_i , v_f , ℓ).

Llamadas repetidas en **ContarCaminos**

Ejercicio

Demuestre que en el peor caso se deben realizar $\frac{n^\ell - 1}{n - 1}$ llamadas al procedimiento **ContarCaminos**, suponiendo que la entrada es $M[1 \dots n][1 \dots n]$, v_i, v_f, ℓ con $n \geq 2$.

- ¿Para qué grafos obtenemos el peor caso?

El algoritmo realiza **un número exponencial de llamadas repetidas**. Dado que sólo podemos tener $n^2 \cdot \ell$ llamadas distintas en la ejecución de **ContarCaminos**($M[1 \dots n][1 \dots n]$, v_i, v_f, ℓ).

Puesto que tenemos **llamadas traslapadas** y un **espacio pequeño de sub-problemas** este es un problema adecuado para **programación dinámica**.

Una tercera definición de **ContarCaminos**

Una tercera definición de **ContarCaminos**

Queremos calcular el número de caminos de largo ℓ desde un nodo v_i a un nodo v_f en un grafo G (representado por una matriz de adyacencia M)

Una tercera definición de **ContarCaminos**

Queremos calcular el número de caminos de largo ℓ desde un nodo v_i a un nodo v_f en un grafo G (representado por una matriz de adyacencia M)

Para evitar hacer llamadas recursivas repetidas, y así disminuir el número de llamadas recursivas, definimos una **secuencia de matrices** H_1, \dots, H_ℓ tales que:

Una tercera definición de **ContarCaminos**

Queremos calcular el número de caminos de largo ℓ desde un nodo v_i a un nodo v_f en un grafo G (representado por una matriz de adyacencia M)

Para evitar hacer llamadas recursivas repetidas, y así disminuir el número de llamadas recursivas, definimos una **secuencia de matrices** H_1, \dots, H_ℓ tales que:

$$H_k[v_i, v_j] \quad := \quad \text{número de caminos de } v_i \text{ a } v_j \text{ de largo } k$$

Una tercera definición de **ContarCaminos**

Queremos calcular el número de caminos de largo ℓ desde un nodo v_i a un nodo v_f en un grafo G (representado por una matriz de adyacencia M)

Para evitar hacer llamadas recursivas repetidas, y así disminuir el número de llamadas recursivas, definimos una **secuencia de matrices** H_1, \dots, H_ℓ tales que:

$$H_k[v_i, v_j] \quad := \quad \text{número de caminos de } v_i \text{ a } v_j \text{ de largo } k$$

La secuencia H_1, \dots, H_ℓ se puede definir recursivamente de la siguiente forma:

Una tercera definición de **ContarCaminos**

Queremos calcular el número de caminos de largo ℓ desde un nodo v_i a un nodo v_f en un grafo G (representado por una matriz de adyacencia M)

Para evitar hacer llamadas recursivas repetidas, y así disminuir el número de llamadas recursivas, definimos una **secuencia de matrices** H_1, \dots, H_ℓ tales que:

$$H_k[v_i, v_j] \quad := \quad \text{número de caminos de } v_i \text{ a } v_j \text{ de largo } k$$

La secuencia H_1, \dots, H_ℓ se puede definir recursivamente de la siguiente forma:

1. $H_1 = M$

Una tercera definición de **ContarCaminos**

Queremos calcular el número de caminos de largo ℓ desde un nodo v_i a un nodo v_f en un grafo G (representado por una matriz de adyacencia M)

Para evitar hacer llamadas recursivas repetidas, y así disminuir el número de llamadas recursivas, definimos una **secuencia de matrices** H_1, \dots, H_ℓ tales que:

$$H_k[v_i, v_j] \quad := \quad \text{número de caminos de } v_i \text{ a } v_j \text{ de largo } k$$

La secuencia H_1, \dots, H_ℓ se puede definir recursivamente de la siguiente forma:

1. $H_1 = M$
2. $H_{k+1} = M \cdot H_k$ para $k \in \{1, \dots, \ell - 1\}$

Una tercera definición de **ContarCaminos**

Queremos calcular el número de caminos de largo ℓ desde un nodo v_i a un nodo v_f en un grafo G (representado por una matriz de adyacencia M)

Para evitar hacer llamadas recursivas repetidas, y así disminuir el número de llamadas recursivas, definimos una **secuencia de matrices** H_1, \dots, H_ℓ tales que:

$$H_k[v_i, v_j] \quad := \quad \text{número de caminos de } v_i \text{ a } v_j \text{ de largo } k$$

La secuencia H_1, \dots, H_ℓ se puede definir recursivamente de la siguiente forma:

1. $H_1 = M$
2. $H_{k+1} = M \cdot H_k$ para $k \in \{1, \dots, \ell - 1\}$

¿Dónde estaría entonces la respuesta a la pregunta original?

Una tercera definición de **ContarCaminos**

Queremos calcular el número de caminos de largo ℓ desde un nodo v_i a un nodo v_f en un grafo G (representado por una matriz de adyacencia M)

Para evitar hacer llamadas recursivas repetidas, y así disminuir el número de llamadas recursivas, definimos una **secuencia de matrices** H_1, \dots, H_ℓ tales que:

$$H_k[v_i, v_j] \quad := \quad \text{número de caminos de } v_i \text{ a } v_j \text{ de largo } k$$

La secuencia H_1, \dots, H_ℓ se puede definir recursivamente de la siguiente forma:

1. $H_1 = M$
2. $H_{k+1} = M \cdot H_k$ para $k \in \{1, \dots, \ell - 1\}$

¿Dónde estaría entonces la respuesta a la pregunta original?

R: en $H_\ell[v_i, v_f]$

Una tercera definición de **ContarCaminos**

La implementación recursiva de la idea descrita:

Una tercera definición de **ContarCaminos**

La implementación recursiva de la idea descrita:

ContarTodosCaminos($M[1 \dots n][1 \dots n]$, ℓ)

if $\ell = 1$ **then return** M

else

$H := \text{ContarTodosCaminos}(M, \ell - 1)$

return $M \cdot H$

Una tercera definición de **ContarCaminos**

La implementación recursiva de la idea descrita:

ContarTodosCaminos($M[1 \dots n][1 \dots n]$, ℓ)

if $\ell = 1$ **then return** M

else

$H := \text{ContarTodosCaminos}(M, \ell - 1)$

return $M \cdot H$

ContarCaminos($M[1 \dots n][1 \dots n]$, v_i , v_f , ℓ)

$H := \text{ContarTodosCaminos}(M, \ell)$

return $H[v_i, v_f]$

La complejidad de **ContarCaminos**

Ejercicio

Demuestre que **ContarCaminos** en el peor caso es $O(\ell \cdot n^3)$

- ¿Cuál es el tamaño de la entrada para **ContarCaminos**?
- ¿Qué operaciones básicas debemos considerar en el análisis de **ContarCaminos**?
- Es necesario utilizar un algoritmo para multiplicar matrices para obtener este resultado

Outline

Programación dinámica: Grafos

Programación dinámica: Palabras

Programación dinámica: un segundo ingrediente

En general, programación dinámica es usada para resolver **problemas de optimización**.

Programación dinámica: un segundo ingrediente

En general, programación dinámica es usada para resolver **problemas de optimización**.

Para que un problema de optimización pueda ser resuelto con esta técnica se debe cumplir el siguiente **principio de optimalidad para los sub-problemas**:

Programación dinámica: un segundo ingrediente

En general, programación dinámica es usada para resolver **problemas de optimización**.

Para que un problema de optimización pueda ser resuelto con esta técnica se debe cumplir el siguiente **principio de optimalidad para los sub-problemas**:

Una solución óptima para un problema contiene soluciones óptimas para sus sub-problemas.

Programación dinámica: un segundo ingrediente

En general, programación dinámica es usada para resolver **problemas de optimización**.

Para que un problema de optimización pueda ser resuelto con esta técnica se debe cumplir el siguiente **principio de optimalidad para los sub-problemas**:

Una solución óptima para un problema contiene soluciones óptimas para sus sub-problemas.

Programación dinámica: un segundo ingrediente

En general, programación dinámica es usada para resolver **problemas de optimización**.

Para que un problema de optimización pueda ser resuelto con esta técnica se debe cumplir el siguiente **principio de optimalidad para los sub-problemas**:

Una solución óptima para un problema contiene soluciones óptimas para sus sub-problemas.

Vamos a ver un ejemplo de este principio que enfatiza otra característica de programación dinámica: en general los problemas deben ser resueltos de forma **bottom-up**.

Midiendo la distancia entre dos palabras

Midiendo la distancia entre dos palabras

Sea $\Sigma = \{a, b, \dots, z\}$ el alfabeto español, el cual contiene 27 símbolos.

Midiendo la distancia entre dos palabras

Sea $\Sigma = \{a, b, \dots, z\}$ el alfabeto español, el cual contiene 27 símbolos.

Vamos a considerar las palabras $w \in \Sigma^*$ (strings) sobre el alfabeto Σ .

Midiendo la distancia entre dos palabras

Sea $\Sigma = \{a, b, \dots, z\}$ el alfabeto español, el cual contiene 27 símbolos.

Vamos a considerar las palabras $w \in \Sigma^*$ (strings) sobre el alfabeto Σ .

Vamos a utilizar la **distancia de Levenshtein** para medir cuán similares son dos palabras.

- Esta es una de las medidas de similitud de palabras más populares por lo que usualmente es llamada **edit distance**.

Midiendo la distancia entre dos palabras

Sea $\Sigma = \{a, b, \dots, z\}$ el alfabeto español, el cual contiene 27 símbolos.

Vamos a considerar las palabras $w \in \Sigma^*$ (strings) sobre el alfabeto Σ .

Vamos a utilizar la **distancia de Levenshtein** para medir cuán similares son dos palabras.

- Esta es una de las medidas de similitud de palabras más populares por lo que usualmente es llamada **edit distance**.

Dadas dos palabras $w_1, w_2 \in \Sigma^*$, utilizamos la notación $\text{ed}(w_1, w_2)$ para la edit distance entre w_1 y w_2

- Tenemos que $\text{ed} : \Sigma^* \times \Sigma^* \rightarrow \mathbb{N}$

Tres operadores sobre palabras

Tres operadores sobre palabras

Sea $w \in \Sigma^*$ con $w = a_1 \cdots a_n$ y $n \geq 0$

- Si $n = 0$ entonces $w = \varepsilon$

Tres operadores sobre palabras

Sea $w \in \Sigma^*$ con $w = a_1 \cdots a_n$ y $n \geq 0$

■ Si $n = 0$ entonces $w = \varepsilon$

Para $i \in \{1, \dots, n\}$ y $b \in \Sigma$ tenemos que:

$$\text{eliminar}(w, i) = a_1 \cdots a_{i-1} a_{i+1} \cdots a_n$$

$$\text{agregar}(w, i, b) = a_1 \cdots a_i b a_{i+1} \cdots a_n$$

$$\text{cambiar}(w, i, b) = a_1 \cdots a_{i-1} b a_{i+1} \cdots a_n$$

Además, tenemos que $\text{agregar}(w, 0, b) = bw$.

Tres operadores sobre palabras

Sea $w \in \Sigma^*$ con $w = a_1 \cdots a_n$ y $n \geq 0$

■ Si $n = 0$ entonces $w = \varepsilon$

Para $i \in \{1, \dots, n\}$ y $b \in \Sigma$ tenemos que:

$$\text{eliminar}(w, i) = a_1 \cdots a_{i-1} a_{i+1} \cdots a_n$$

$$\text{agregar}(w, i, b) = a_1 \cdots a_i b a_{i+1} \cdots a_n$$

$$\text{cambiar}(w, i, b) = a_1 \cdots a_{i-1} b a_{i+1} \cdots a_n$$

Además, tenemos que $\text{agregar}(w, 0, b) = bw$.

Ejemplo

Tres operadores sobre palabras

Sea $w \in \Sigma^*$ con $w = a_1 \cdots a_n$ y $n \geq 0$

■ Si $n = 0$ entonces $w = \varepsilon$

Para $i \in \{1, \dots, n\}$ y $b \in \Sigma$ tenemos que:

$$\text{eliminar}(w, i) = a_1 \cdots a_{i-1} a_{i+1} \cdots a_n$$

$$\text{agregar}(w, i, b) = a_1 \cdots a_i b a_{i+1} \cdots a_n$$

$$\text{cambiar}(w, i, b) = a_1 \cdots a_{i-1} b a_{i+1} \cdots a_n$$

Además, tenemos que $\text{agregar}(w, 0, b) = bw$.

Ejemplo

$$\text{eliminar}(\text{hola}, 1) = \text{ola}$$

Tres operadores sobre palabras

Sea $w \in \Sigma^*$ con $w = a_1 \cdots a_n$ y $n \geq 0$

■ Si $n = 0$ entonces $w = \varepsilon$

Para $i \in \{1, \dots, n\}$ y $b \in \Sigma$ tenemos que:

$$\text{eliminar}(w, i) = a_1 \cdots a_{i-1} a_{i+1} \cdots a_n$$

$$\text{agregar}(w, i, b) = a_1 \cdots a_i b a_{i+1} \cdots a_n$$

$$\text{cambiar}(w, i, b) = a_1 \cdots a_{i-1} b a_{i+1} \cdots a_n$$

Además, tenemos que $\text{agregar}(w, 0, b) = bw$.

Ejemplo

$$\text{eliminar}(\text{hola}, 1) = \text{ola}$$

$$\text{eliminar}(\text{hola}, 3) = \text{hoa}$$

Tres operadores sobre palabras

Sea $w \in \Sigma^*$ con $w = a_1 \cdots a_n$ y $n \geq 0$

■ Si $n = 0$ entonces $w = \varepsilon$

Para $i \in \{1, \dots, n\}$ y $b \in \Sigma$ tenemos que:

$$\text{eliminar}(w, i) = a_1 \cdots a_{i-1} a_{i+1} \cdots a_n$$

$$\text{agregar}(w, i, b) = a_1 \cdots a_i b a_{i+1} \cdots a_n$$

$$\text{cambiar}(w, i, b) = a_1 \cdots a_{i-1} b a_{i+1} \cdots a_n$$

Además, tenemos que $\text{agregar}(w, 0, b) = bw$.

Ejemplo

$$\begin{array}{lll} \text{eliminar}(\text{hola}, 1) & = & \text{ola} \\ \text{agregar}(\text{hola}, 0, y) & = & y\text{hola} \end{array} \qquad \text{eliminar}(\text{hola}, 3) = \text{hoa}$$

Tres operadores sobre palabras

Sea $w \in \Sigma^*$ con $w = a_1 \cdots a_n$ y $n \geq 0$

■ Si $n = 0$ entonces $w = \varepsilon$

Para $i \in \{1, \dots, n\}$ y $b \in \Sigma$ tenemos que:

$$\text{eliminar}(w, i) = a_1 \cdots a_{i-1} a_{i+1} \cdots a_n$$

$$\text{agregar}(w, i, b) = a_1 \cdots a_i b a_{i+1} \cdots a_n$$

$$\text{cambiar}(w, i, b) = a_1 \cdots a_{i-1} b a_{i+1} \cdots a_n$$

Además, tenemos que $\text{agregar}(w, 0, b) = bw$.

Ejemplo

$\text{eliminar}(\text{hola}, 1)$	$=$	ola	$\text{eliminar}(\text{hola}, 3)$	$=$	hoa
$\text{agregar}(\text{hola}, 0, y)$	$=$	yhola	$\text{agregar}(\text{hola}, 3, z)$	$=$	holza

Tres operadores sobre palabras

Sea $w \in \Sigma^*$ con $w = a_1 \cdots a_n$ y $n \geq 0$

■ Si $n = 0$ entonces $w = \varepsilon$

Para $i \in \{1, \dots, n\}$ y $b \in \Sigma$ tenemos que:

$$\text{eliminar}(w, i) = a_1 \cdots a_{i-1} a_{i+1} \cdots a_n$$

$$\text{agregar}(w, i, b) = a_1 \cdots a_i b a_{i+1} \cdots a_n$$

$$\text{cambiar}(w, i, b) = a_1 \cdots a_{i-1} b a_{i+1} \cdots a_n$$

Además, tenemos que $\text{agregar}(w, 0, b) = bw$.

Ejemplo

$\text{eliminar}(\text{hola}, 1)$	$=$	ola	$\text{eliminar}(\text{hola}, 3)$	$=$	hoa
$\text{agregar}(\text{hola}, 0, y)$	$=$	yhola	$\text{agregar}(\text{hola}, 3, z)$	$=$	holza
$\text{cambiar}(\text{hola}, 2, x)$	$=$	hxla			

La distancia de Levenshtein

Dadas palabras $w_1, w_2 \in \Sigma^*$, definimos $ed(w_1, w_2)$ como el menor número de operaciones eliminar, agregar y cambiar que aplicadas desde w_1 generan w_2

Ejemplo

Tenemos que $ed(\text{casa}, \text{asado}) = 3$ puesto que:

agregar(casa, 4, d) = casad

eliminar(casad, 1) = asad

agregar(asad, 4, o) = asado

La distancia de Levenshtein

Ejercicios

1. Demuestre que $\text{ed}(w_1, w_2) \leq |w_1| + |w_2|$
2. Demuestre que ed es una función de distancia, vale decir, muestre lo siguiente:
 - 2.1 $\text{ed}(w_1, w_2) = 0$ si y sólo si $w_1 = w_2$
 - 2.2 $\text{ed}(w_1, w_2) = \text{ed}(w_2, w_1)$
 - 2.3 $\text{ed}(w_1, w_2) \leq \text{ed}(w_1, w_3) + \text{ed}(w_3, w_2)$
3. Dadas dos palabras w_1, w_2 del mismo largo, la distancia de Hamming entre w_1 y w_2 es definida como el número de posiciones en las que tienen distintos símbolos. Denote esta distancia como $\text{hd}(w_1, w_2)$
 - 3.1 Demuestre que $\text{ed}(w_1, w_2) \leq \text{hd}(w_1, w_2)$
 - 3.2 Encuentre palabras w_3 y w_4 tales que $\text{ed}(w_3, w_4) < \text{hd}(w_3, w_4)$

Calculando la distancia de Levenshtein

Para calcular $ed(w_1, w_2)$ no podemos considerar todas las posibles secuencias de operaciones que aplicadas desde w_1 generan w_2 .

Calculando la distancia de Levenshtein

Para calcular $ed(w_1, w_2)$ no podemos considerar todas las posibles secuencias de operaciones que aplicadas desde w_1 generan w_2 .

- Incluso si consideramos las secuencias de largo a los más $|w_1| + |w_2|$ vamos a tener demasiadas posibilidades.

Calculando la distancia de Levenshtein

Para calcular $ed(w_1, w_2)$ no podemos considerar todas las posibles secuencias de operaciones que aplicadas desde w_1 generan w_2 .

- Incluso si consideramos las secuencias de largo a los más $|w_1| + |w_2|$ vamos a tener demasiadas posibilidades.

Para calcular $ed(w_1, w_2)$ utilizamos **programación dinámica**.

Notación preliminar

Notación preliminar

Dado $w \in \Sigma^*$ tal que $|w| = n$, y dado $i \in \{1, \dots, n\}$, definimos $w[i]$ como el símbolo de w en la posición i .

Notación preliminar

Dado $w \in \Sigma^*$ tal que $|w| = n$, y dado $i \in \{1, \dots, n\}$, definimos $w[i]$ como el símbolo de w en la posición i .

- Tenemos que $w = w[1] \cdots w[n]$.

Notación preliminar

Dado $w \in \Sigma^*$ tal que $|w| = n$, y dado $i \in \{1, \dots, n\}$, definimos $w[i]$ como el símbolo de w en la posición i .

- Tenemos que $w = w[1] \cdots w[n]$.

Además, definimos el **infijo** (*substring*) de w entre las posiciones i y j como:

$$w[i, j] = \begin{cases} w[i] \cdots w[j] & 1 \leq i \leq j \leq n \\ \varepsilon & \text{en caso contrario} \end{cases}$$

La distancia y un principio optimalidad para sub-secuencias

Sean $w_1, w_2 \in \Sigma^*$.

La distancia y un principio optimalidad para sub-secuencias

Sean $w_1, w_2 \in \Sigma^*$.

Suponga que o_1, \dots, o_k es una secuencia óptima de operaciones que aplicadas desde w_1 generan w_2 con $k \geq 1$.

La distancia y un principio optimalidad para sub-secuencias

Sean $w_1, w_2 \in \Sigma^*$.

Suponga que o_1, \dots, o_k es una secuencia óptima de operaciones que aplicadas desde w_1 generan w_2 con $k \geq 1$.

- Tenemos que $\text{ed}(w_1, w_2) = k$.

La distancia y un principio optimalidad para sub-secuencias

Sean $w_1, w_2 \in \Sigma^*$.

Suponga que o_1, \dots, o_k es una secuencia óptima de operaciones que aplicadas desde w_1 generan w_2 con $k \geq 1$.

- Tenemos que $\text{ed}(w_1, w_2) = k$.

Considere $0 \leq i \leq |w_1|$ y suponga que o_1, \dots, o_ℓ es la sub-secuencia de las operaciones en o_1, \dots, o_k que son aplicadas a $w_1[1, i]$ con $1 \leq \ell \leq k$.

La distancia y un principio optimalidad para sub-secuencias

Sean $w_1, w_2 \in \Sigma^*$.

Suponga que o_1, \dots, o_k es una secuencia óptima de operaciones que aplicadas desde w_1 generan w_2 con $k \geq 1$.

- Tenemos que $\text{ed}(w_1, w_2) = k$.

Considere $0 \leq i \leq |w_1|$ y suponga que o_1, \dots, o_ℓ es la sub-secuencia de las operaciones en o_1, \dots, o_k que son aplicadas a $w_1[1, i]$ con $1 \leq \ell \leq k$.

Estamos suponiendo que las operaciones sobre $w_1[1, i]$ son las **primeras en ser aplicadas**.

La distancia y un principio optimalidad para sub-secuencias

Sean $w_1, w_2 \in \Sigma^*$.

Suponga que o_1, \dots, o_k es una secuencia óptima de operaciones que aplicadas desde w_1 generan w_2 con $k \geq 1$.

- Tenemos que $\text{ed}(w_1, w_2) = k$.

Considere $0 \leq i \leq |w_1|$ y suponga que o_1, \dots, o_ℓ es la sub-secuencia de las operaciones en o_1, \dots, o_k que son aplicadas a $w_1[1, i]$ con $1 \leq \ell \leq k$.

Estamos suponiendo que las operaciones sobre $w_1[1, i]$ son las **primeras en ser aplicadas**.

¿Por qué podemos hacer este supuesto?

La distancia y un principio optimalidad para sub-secuencias

Podemos realizar el supuesto por la siguiente razón:

La distancia y un principio optimalidad para sub-secuencias

Podemos realizar el supuesto por la siguiente razón: si para $s \in \{1, \dots, \ell - 1\}$ tenemos que o_{s+1} es una operación sobre $w_1[1, i]$ pero o_s no lo es, entonces podemos **permutar** estas dos operaciones para generar o_{s+1}, o_s^* tal que:

La distancia y un principio optimalidad para sub-secuencias

Podemos realizar el supuesto por la siguiente razón: si para $s \in \{1, \dots, \ell - 1\}$ tenemos que o_{s+1} es una operación sobre $w_1[1, i]$ pero o_s no lo es, entonces podemos **permutar** estas dos operaciones para generar o_{s+1}, o_s^* tal que:

- si o_{s+1} **cambia** un símbolo por otro, entonces $o_s^* = o_s$.

La distancia y un principio optimalidad para sub-secuencias

Podemos realizar el supuesto por la siguiente razón: si para $s \in \{1, \dots, \ell - 1\}$ tenemos que o_{s+1} es una operación sobre $w_1[1, i]$ pero o_s no lo es, entonces podemos **permutar** estas dos operaciones para generar o_{s+1}, o_s^* tal que:

- si o_{s+1} **cambia** un símbolo por otro, entonces $o_s^* = o_s$.
- si o_{s+1} **agrega** un símbolo, entonces o_s^* se obtiene desde o_s cambiando la posición $t \in \{1, \dots, n\}$ mencionada en o_s por $t + 1$.

La distancia y un principio optimalidad para sub-secuencias

Podemos realizar el supuesto por la siguiente razón: si para $s \in \{1, \dots, \ell - 1\}$ tenemos que o_{s+1} es una operación sobre $w_1[1, i]$ pero o_s no lo es, entonces podemos **permutar** estas dos operaciones para generar o_{s+1}, o_s^* tal que:

- si o_{s+1} **cambia** un símbolo por otro, entonces $o_s^* = o_s$.
- si o_{s+1} **agrega** un símbolo, entonces o_s^* se obtiene desde o_s cambiando la posición $t \in \{1, \dots, n\}$ mencionada en o_s por $t + 1$.
- si o_{s+1} **elimina** un símbolo, entonces o_s^* se obtiene desde o_s cambiando la posición $t \in \{1, \dots, n\}$ mencionada en o_s por $t - 1$.

La distancia y un principio optimalidad para sub-secuencias

Podemos realizar el supuesto por la siguiente razón: si para $s \in \{1, \dots, \ell - 1\}$ tenemos que o_{s+1} es una operación sobre $w_1[1, i]$ pero o_s no lo es, entonces podemos **permutar** estas dos operaciones para generar o_{s+1}, o_s^* tal que:

- si o_{s+1} **cambia** un símbolo por otro, entonces $o_s^* = o_s$.
- si o_{s+1} **agrega** un símbolo, entonces o_s^* se obtiene desde o_s cambiando la posición $t \in \{1, \dots, n\}$ mencionada en o_s por $t + 1$.
- si o_{s+1} **elimina** un símbolo, entonces o_s^* se obtiene desde o_s cambiando la posición $t \in \{1, \dots, n\}$ mencionada en o_s por $t - 1$.

Finalmente, suponga que la aplicación de o_1, \dots, o_ℓ desde $w_1[1, i]$ genera $w_2[1, j]$ con $0 \leq j \leq |w_2|$

La distancia y un principio optimalidad para sub-secuencias

Entonces la sub-secuencia o_1, \dots, o_ℓ debe ser óptima para la generación de $w_2[1, j]$ a partir de $w_1[1, i]$.

La distancia y un principio optimalidad para sub-secuencias

Entonces la sub-secuencia o_1, \dots, o_ℓ debe ser óptima para la generación de $w_2[1, j]$ a partir de $w_1[1, i]$.

- Vale decir, $\text{ed}(w_1[1, i], w_2[1, j]) = \ell$.

La distancia y un principio optimalidad para sub-secuencias

Entonces la sub-secuencia o_1, \dots, o_ℓ debe ser óptima para la generación de $w_2[1, j]$ a partir de $w_1[1, i]$.

- Vale decir, $\text{ed}(w_1[1, i], w_2[1, j]) = \ell$.

Demostración

Suponga que lo anterior no es cierto, y suponga que o'_1, \dots, o'_m es una secuencia de operaciones con $m < \ell$ que genera $w_2[1, j]$ desde $w_1[1, i]$

La distancia y un principio optimalidad para sub-secuencias

Entonces la sub-secuencia o_1, \dots, o_ℓ debe ser óptima para la generación de $w_2[1, j]$ a partir de $w_1[1, i]$.

- Vale decir, $\text{ed}(w_1[1, i], w_2[1, j]) = \ell$.

Demostración

Suponga que lo anterior no es cierto, y suponga que o'_1, \dots, o'_m es una secuencia de operaciones con $m < \ell$ que genera $w_2[1, j]$ desde $w_1[1, i]$

En la secuencia o_1, \dots, o_ℓ que genera w_2 desde w_1 podemos reemplazar o_1, \dots, o_ℓ por o'_1, \dots, o'_m

- La secuencia resultante se puede utilizar para generar w_2 desde w_1

La distancia y un principio optimalidad para sub-secuencias

Entonces la sub-secuencia o_1, \dots, o_ℓ debe ser óptima para la generación de $w_2[1, j]$ a partir de $w_1[1, i]$.

- Vale decir, $\text{ed}(w_1[1, i], w_2[1, j]) = \ell$.

Demostración

Suponga que lo anterior no es cierto, y suponga que o'_1, \dots, o'_m es una secuencia de operaciones con $m < \ell$ que genera $w_2[1, j]$ desde $w_1[1, i]$

En la secuencia o_1, \dots, o_k que genera w_2 desde w_1 podemos reemplazar o_1, \dots, o_ℓ por o'_1, \dots, o'_m

- La secuencia resultante se puede utilizar para generar w_2 desde w_1

Concluimos que $\text{ed}(w_1, w_2) \leq (k - \ell) + m$

La distancia y un principio optimalidad para sub-secuencias

Entonces la sub-secuencia o_1, \dots, o_ℓ debe ser óptima para la generación de $w_2[1, j]$ a partir de $w_1[1, i]$.

- Vale decir, $\text{ed}(w_1[1, i], w_2[1, j]) = \ell$.

Demostración

Suponga que lo anterior no es cierto, y suponga que o'_1, \dots, o'_m es una secuencia de operaciones con $m < \ell$ que genera $w_2[1, j]$ desde $w_1[1, i]$

En la secuencia o_1, \dots, o_k que genera w_2 desde w_1 podemos reemplazar o_1, \dots, o_ℓ por o'_1, \dots, o'_m

- La secuencia resultante se puede utilizar para generar w_2 desde w_1

Concluimos que $\text{ed}(w_1, w_2) \leq (k - \ell) + m = k - (\ell - m)$

La distancia y un principio optimalidad para sub-secuencias

Entonces la sub-secuencia o_1, \dots, o_ℓ debe ser óptima para la generación de $w_2[1, j]$ a partir de $w_1[1, i]$.

- Vale decir, $\text{ed}(w_1[1, i], w_2[1, j]) = \ell$.

Demostración

Suponga que lo anterior no es cierto, y suponga que o'_1, \dots, o'_m es una secuencia de operaciones con $m < \ell$ que genera $w_2[1, j]$ desde $w_1[1, i]$

En la secuencia o_1, \dots, o_k que genera w_2 desde w_1 podemos reemplazar o_1, \dots, o_ℓ por o'_1, \dots, o'_m

- La secuencia resultante se puede utilizar para generar w_2 desde w_1

Concluimos que $\text{ed}(w_1, w_2) \leq (k - \ell) + m = k - (\ell - m) < k$

La distancia y un principio optimalidad para sub-secuencias

Entonces la sub-secuencia o_1, \dots, o_ℓ debe ser óptima para la generación de $w_2[1, j]$ a partir de $w_1[1, i]$.

- Vale decir, $\text{ed}(w_1[1, i], w_2[1, j]) = \ell$.

Demostración

Suponga que lo anterior no es cierto, y suponga que o'_1, \dots, o'_m es una secuencia de operaciones con $m < \ell$ que genera $w_2[1, j]$ desde $w_1[1, i]$

En la secuencia o_1, \dots, o_k que genera w_2 desde w_1 podemos reemplazar o_1, \dots, o_ℓ por o'_1, \dots, o'_m

- La secuencia resultante se puede utilizar para generar w_2 desde w_1

Concluimos que $\text{ed}(w_1, w_2) \leq (k - \ell) + m = k - (\ell - m) < k$, lo que contradice el supuesto inicial.

La distancia y un principio optimalidad para sub-secuencias

Entonces la sub-secuencia o_1, \dots, o_ℓ debe ser óptima para la generación de $w_2[1, j]$ a partir de $w_1[1, i]$.

- Vale decir, $\text{ed}(w_1[1, i], w_2[1, j]) = \ell$.

Demostración

Suponga que lo anterior no es cierto, y suponga que o'_1, \dots, o'_m es una secuencia de operaciones con $m < \ell$ que genera $w_2[1, j]$ desde $w_1[1, i]$

En la secuencia o_1, \dots, o_k que genera w_2 desde w_1 podemos reemplazar o_1, \dots, o_ℓ por o'_1, \dots, o'_m

- La secuencia resultante se puede utilizar para generar w_2 desde w_1

Concluimos que $\text{ed}(w_1, w_2) \leq (k - \ell) + m = k - (\ell - m) < k$, lo que contradice el supuesto inicial.



Una definición recursiva de la distancia de Levenshtein

Una definición recursiva de la distancia de Levenshtein

Fije dos strings $w_1, w_2 \in \Sigma^*$ tales que $|w_1| = m$ y $|w_2| = n$.

Una definición recursiva de la distancia de Levenshtein

Fije dos strings $w_1, w_2 \in \Sigma^*$ tales que $|w_1| = m$ y $|w_2| = n$.

Dados $i \in \{0, \dots, m\}$ y $j \in \{0, \dots, n\}$, definimos

Una definición recursiva de la distancia de Levenshtein

Fije dos strings $w_1, w_2 \in \Sigma^*$ tales que $|w_1| = m$ y $|w_2| = n$.

Dados $i \in \{0, \dots, m\}$ y $j \in \{0, \dots, n\}$, definimos

$$\text{ed}(i, j) = \text{ed}(w_1[1, i], w_2[1, j])$$

Una definición recursiva de la distancia de Levenshtein

Fije dos strings $w_1, w_2 \in \Sigma^*$ tales que $|w_1| = m$ y $|w_2| = n$.

Dados $i \in \{0, \dots, m\}$ y $j \in \{0, \dots, n\}$, definimos

$$\text{ed}(i, j) = \text{ed}(w_1[1, i], w_2[1, j])$$

Observe que $\text{ed}(w_1, w_2) = \text{ed}(m, n)$

Una definición recursiva de la distancia de Levenshtein

Fije dos strings $w_1, w_2 \in \Sigma^*$ tales que $|w_1| = m$ y $|w_2| = n$.

Dados $i \in \{0, \dots, m\}$ y $j \in \{0, \dots, n\}$, definimos

$$\text{ed}(i, j) = \text{ed}(w_1[1, i], w_2[1, j])$$

Observe que $\text{ed}(w_1, w_2) = \text{ed}(m, n)$

Además, definimos el valor $\text{dif}(i, j)$ como 0 si $w_1[i] = w_2[j]$, y como 1 en caso contrario.

Una definición recursiva de la distancia de Levenshtein

Del principio de optimalidad para sub-secuencias obtenemos la siguiente **definición recursiva** para la función ed :

Una definición recursiva de la distancia de Levenshtein

Del principio de optimalidad para sub-secuencias obtenemos la siguiente **definición recursiva** para la función ed :

$$ed(i, j) = \begin{cases} \max\{i, j\} & i = 0 \text{ o } j = 0 \\ \end{cases}$$

Una definición recursiva de la distancia de Levenshtein

Del principio de optimalidad para sub-secuencias obtenemos la siguiente **definición recursiva** para la función ed :

$$ed(i, j) = \begin{cases} \max\{i, j\} & i = 0 \text{ o } j = 0 \\ \min\{1 + ed(i - 1, j), \\ \quad 1 + ed(i, j - 1), \\ \quad \text{dif}(i, j) + ed(i - 1, j - 1)\} & i > 0 \text{ y } j > 0 \end{cases}$$

Una definición recursiva de la distancia de Levenshtein

Del principio de optimalidad para sub-secuencias obtenemos la siguiente **definición recursiva** para la función ed :

$$ed(i, j) = \begin{cases} \max\{i, j\} & i = 0 \text{ o } j = 0 \\ \min\{1 + ed(i - 1, j), \\ \quad 1 + ed(i, j - 1), \\ \quad \text{dif}(i, j) + ed(i - 1, j - 1)\} & i > 0 \text{ y } j > 0 \end{cases}$$

Tenemos entonces una forma de calcular la función ed que se basa en resolver sub-problemas más pequeños

Una definición recursiva de la distancia de Levenshtein

Del principio de optimalidad para sub-secuencias obtenemos la siguiente **definición recursiva** para la función ed :

$$ed(i, j) = \begin{cases} \max\{i, j\} & i = 0 \text{ o } j = 0 \\ \min\{1 + ed(i - 1, j), \\ \quad 1 + ed(i, j - 1), \\ \quad \text{dif}(i, j) + ed(i - 1, j - 1)\} & i > 0 \text{ y } j > 0 \end{cases}$$

Tenemos entonces una forma de calcular la función ed que se basa en resolver sub-problemas más pequeños

- Estos sub-problemas están traslapados y se tiene un número polinomial de ellos, podemos aplicar entonces programación dinámica

Una implementación recursiva de la distancia de Levenshtein

EditDistance(w_1 , i , w_2 , j)

Una implementación recursiva de la distancia de Levenshtein

```
EditDistance( $w_1$ ,  $i$ ,  $w_2$ ,  $j$ )  
  if  $i = 0$  then return  $j$   
  else if  $j = 0$  then return  $i$ 
```

Una implementación recursiva de la distancia de Levenshtein

```
EditDistance( $w_1, i, w_2, j$ )  
  if  $i = 0$  then return  $j$   
  else if  $j = 0$  then return  $i$   
  else  
     $r :=$  EditDistance( $w_1, i - 1, w_2, j$ )  
     $s :=$  EditDistance( $w_1, i, w_2, j - 1$ )  
     $t :=$  EditDistance( $w_1, i - 1, w_2, j - 1$ )
```

Una implementación recursiva de la distancia de Levenshtein

```
EditDistance( $w_1$ ,  $i$ ,  $w_2$ ,  $j$ )  
  if  $i = 0$  then return  $j$   
  else if  $j = 0$  then return  $i$   
  else  
     $r :=$  EditDistance( $w_1$ ,  $i - 1$ ,  $w_2$ ,  $j$ )  
     $s :=$  EditDistance( $w_1$ ,  $i$ ,  $w_2$ ,  $j - 1$ )  
     $t :=$  EditDistance( $w_1$ ,  $i - 1$ ,  $w_2$ ,  $j - 1$ )  
    if  $w_1[i] = w_2[j]$  then  $d := 0$   
    else  $d := 1$ 
```

Una implementación recursiva de la distancia de Levenshtein

```
EditDistance( $w_1, i, w_2, j$ )  
  if  $i = 0$  then return  $j$   
  else if  $j = 0$  then return  $i$   
  else  
     $r := \text{EditDistance}(w_1, i - 1, w_2, j)$   
     $s := \text{EditDistance}(w_1, i, w_2, j - 1)$   
     $t := \text{EditDistance}(w_1, i - 1, w_2, j - 1)$   
    if  $w_1[i] = w_2[j]$  then  $d := 0$   
    else  $d := 1$   
    return  $\min\{1 + r, 1 + s, d + t\}$ 
```

Una implementación recursiva de la distancia de Levenshtein

```
EditDistance( $w_1, i, w_2, j$ )  
  if  $i = 0$  then return  $j$   
  else if  $j = 0$  then return  $i$   
  else  
     $r := \text{EditDistance}(w_1, i - 1, w_2, j)$   
     $s := \text{EditDistance}(w_1, i, w_2, j - 1)$   
     $t := \text{EditDistance}(w_1, i - 1, w_2, j - 1)$   
    if  $w_1[i] = w_2[j]$  then  $d := 0$   
    else  $d := 1$   
    return  $\min\{1 + r, 1 + s, d + t\}$ 
```

¿Es esta una buena implementación de **EditDistance**?

Una implementación recursiva de la distancia de Levenshtein

```
EditDistance( $w_1, i, w_2, j$ )  
  if  $i = 0$  then return  $j$   
  else if  $j = 0$  then return  $i$   
  else  
     $r := \text{EditDistance}(w_1, i - 1, w_2, j)$   
     $s := \text{EditDistance}(w_1, i, w_2, j - 1)$   
     $t := \text{EditDistance}(w_1, i - 1, w_2, j - 1)$   
    if  $w_1[i] = w_2[j]$  then  $d := 0$   
    else  $d := 1$   
    return  $\min\{1 + r, 1 + s, d + t\}$ 
```

¿Es esta una buena implementación de **EditDistance**?

R: No porque tenemos muchas llamadas recursivas repetidas, es mejor evaluar esta función utilizando un enfoque bottom-up.

Una evaluación bottom-up de la distancia de Levenshtein

Para determinar los valores de la función ed construimos una tabla siguiendo un orden lexicográfico para los pares (i, j) :

Una evaluación bottom-up de la distancia de Levenshtein

Para determinar los valores de la función ed construimos una tabla siguiendo un orden lexicográfico para los pares (i, j) :

$$(i_1, j_1) < (i_2, j_2) \quad \text{si y sólo si} \quad i_1 < i_2 \text{ o } (i_1 = i_2 \text{ y } j_1 < j_2)$$

Una evaluación bottom-up de la distancia de Levenshtein

Para determinar los valores de la función ed construimos una tabla siguiendo un orden lexicográfico para los pares (i, j) :

$$(i_1, j_1) < (i_2, j_2) \quad \text{si y sólo si} \quad i_1 < i_2 \text{ o } (i_1 = i_2 \text{ y } j_1 < j_2)$$

Por ejemplo, para determinar el valor de $ed(\text{casa}, \text{asado})$ construimos la siguiente tabla:

Una evaluación bottom-up de la distancia de Levenshtein

Para determinar los valores de la función ed construimos una tabla siguiendo un orden lexicográfico para los pares (i, j) :

$$(i_1, j_1) < (i_2, j_2) \quad \text{si y sólo si} \quad i_1 < i_2 \text{ o } (i_1 = i_2 \text{ y } j_1 < j_2)$$

Por ejemplo, para determinar el valor de $ed(\text{casa}, \text{asado})$ construimos la siguiente tabla:

		a	s	a	d	o
c a s a	0	1	2	3	4	5
	1	1	2	3	4	5
	2	1	2	2	3	4
	3	2	1	2	3	4
	4	3	2	1	2	3

Una evaluación bottom-up de la distancia de Levenshtein

A partir de la tabla podemos obtener una secuencia óptima de operaciones para transformar casa en asado:

Una evaluación bottom-up de la distancia de Levenshtein

A partir de la tabla podemos obtener una secuencia óptima de operaciones para transformar casa en asado:

		a	s	a	d	o
	0	1	2	3	4	5
c	1	1	2	3	4	5
a	2	1	2	2	3	4
s	3	2	1	2	3	4
a	4	3	2	1	2	3

Una evaluación bottom-up de la distancia de Levenshtein

A partir de la tabla podemos obtener una secuencia óptima de operaciones para transformar casa en asado:

		a	s	a	d	o	
		0	1	2	3	4	5
c		0	1	2	3	4	5
a		1	1	2	3	4	5
s		2	1	2	2	3	4
a		3	2	1	2	3	4
		4	3	2	1	2	3

Estas operaciones son las siguientes:

eliminar(casa, 1) = asa

agregar(asa, 3, d) = asad

agregar(asad, 4, o) = asado

Una evaluación bottom-up de la distancia de Levenshtein

A partir de la tabla podemos obtener una secuencia óptima de operaciones para transformar casa en asado:

		a	s	a	d	o	
		0	1	2	3	4	5
c		0	1	2	3	4	5
a		1	1	2	3	4	5
s		2	1	2	2	3	4
a		3	2	1	2	3	4
		4	3	2	1	2	3

Estas operaciones son las siguientes:

eliminar(casa, 1) = asa

agregar(asa, 3, d) = asad

agregar(asad, 4, o) = asado

¿Qué operaciones debemos contar al analizar la complejidad del algoritmo discutido en las transparencias anteriores?

El costo de calcular la distancia de Levenshtein

Consideramos como operaciones básicas acceder a una celda de la tabla (para leer o escribir), realizar operaciones con elementos de la tabla (sumar o comparar) y comparar elementos de las palabras de entrada.

El costo de calcular la distancia de Levenshtein

Consideramos como operaciones básicas acceder a una celda de la tabla (para leer o escribir), realizar operaciones con elementos de la tabla (sumar o comparar) y comparar elementos de las palabras de entrada.

Corolario

Utilizando programación dinámica es posible construir un algoritmo para calcular $ed(w_1, w_2)$ que en el peor caso es $\Theta(|w_1| \cdot |w_2|)$