Técnicas Fundamentales

Algoritmo de Karatsuba y Programación Dinámica

Segundo semestre 2022

IIC2283

Prof. Nicolás Van Sint Jan

Recordatorio: Dividir para conquistar

Esta es la forma genérica de una algoritmo que utiliza la técnica de **dividir para conquistar**:

Bosquejo de general de algoritmo

DividirParaConquistar(w)
    if $|w| \leq k$ **then return InstanciasPequeñas**(w)
    else
        Dividir w en $w_1, \ldots, w_\ell$
        for $i := 1$ **to** ℓ **do**
            $S_i :=$ **DividirParaConquistar**(w_i)
        return Combinar($S_1, \ldots, S_\ell$)

¿Cuál es la complejidad de un algoritmo de **dividir para conquistar**?

Recordatorio: Suma de números enteros

Sean $a, b \in \mathbb{Z}$ con $n \geq 1$ dígitos cada uno. Sea

$$c = a + b.$$

Considere **el algoritmo usual de la suma** para calcular c.

Consideramos la **suma de dos dígitos**, **comparación de dos dígitos** y **resta de un número con a lo más dos dígitos con uno de un dígito** como las operaciones a contar, cada una con costo 1.

Preguntas

1. ¿Cuántas operaciones realiza el algoritmo en el peor caso?
2. ¿Cuántos dígitos puede tener c?

¿Se puede **sumar** más rápido que $\mathcal{O}(n)$?

Recordatorio: Multiplicación de números enteros

Sean $a, b \in \mathbb{Z}$ con $n \geq 1$ dígitos cada uno. Sea

$$d = a \cdot b$$

Considere **el algoritmo usual de la multiplicación** para calcular d.

Esta vez tome **la suma** y **la multiplicación de dígitos** como las operaciones a contar, ambas con costo 1.

Preguntas

1. ¿Cuántas operaciones realiza el algoritmo en este caso?
2. ¿Cuántos dígitos puede tener d?

¿Se puede **multiplicar** más rápido que $\mathcal{O}(n^2)$?

Outline

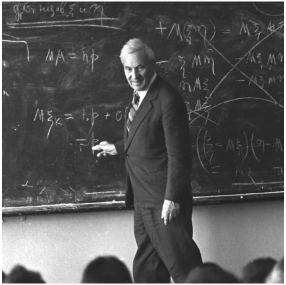
Dividir para conquistar: Multiplicar

Programación dinámica: Grafos

Programación dinámica: Palabras

Outline

Algoritmo de multiplicación de Karatsuba



Andréi Kolmogorov

Anatoli Karatsuba

Algoritmo de multiplicación de Karatsuba

Sean $a, b \in \mathbb{Z}$ con n dígitos cada uno, donde $n = 2^k$ para algún $k \in \mathbb{N}$.

Se puede representar a y b de la siguiente forma:

$$a = a_1 \cdot 10^{\frac{n}{2}} + a_2$$
$$b = b_1 \cdot 10^{\frac{n}{2}} + b_2$$

Tenemos entonces que:

$$a \cdot b = a_1 \cdot b_1 \cdot 10^n + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{\frac{n}{2}} + a_2 \cdot b_2$$

Algoritmo de multiplicación de Karatsuba

$$a \cdot b = a_1 \cdot b_1 \cdot 10^n + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{\frac{n}{2}} + a_2 \cdot b_2$$

Para calcular $a \cdot b$ entonces debemos calcular las siguientes multiplicaciones:

1. $a_1 \cdot b_1$
2. $a_1 \cdot b_2$
3. $a_2 \cdot b_1$
4. $a_2 \cdot b_2$

Obtenemos entonces un algoritmo recursivo

- Para resolver el caso de largo n realizamos 4 llamadas para los casos de largo $\frac{n}{2}$

¿Cuál el la complejidad de este algoritmo?

R: Se puede usar el teorema Maestro para deducir que este algoritmo es de orden $\Theta(n^2)$.

La idea clave en el algoritmo de Karatsuba

Podemos calcular $a \cdot b$ realizando las siguientes multiplicaciones:

1. $c_1 = a_1 \cdot b_1$
2. $c_2 = a_2 \cdot b_2$
3. $c_3 = (a_1 + a_2) \cdot (b_1 + b_2)$

Tenemos entonces que:

$$\boxed{a \cdot b \;=\; c_1 \cdot 10^n + (c_3 - (c_1 + c_2)) \cdot 10^{\frac{n}{2}} + c_2}$$

Esta expresión se conoce como **el algoritmo de Karatsuba**.

¿Cuántas **operaciones** realiza este algoritmo?

Tiempo de ejecución del algoritmo de Karatsuba

Sea $T(n)$ el número de operaciones realizadas en el **peor caso** por el algoritmo de Karatsuba para dos números de entrada con n dígitos cada uno.

Para determinar el orden de $T(n)$ utilizamos la siguiente **ecuación de recurrencia** (con $e \in \mathbb{N}$ una constante):

$$T(n) = \begin{cases} 1 & n = 1 \\ 3 \cdot T\left(\frac{n}{2}\right) + e \cdot n & n > 1 \end{cases}$$

Tiempo de ejecución del algoritmo de Karatsuba

$$a \cdot b \;=\; c_1 \cdot 10^n + (c_3 - (c_1 + c_2)) \cdot 10^{\frac{n}{2}} + c_2$$

$$T(n) \quad = \quad \begin{cases} 1 & n = 1 \\ 3 \cdot T\left(\frac{n}{2}\right) + e \cdot n & n > 1 \end{cases}$$

Preguntas

1. ¿Qué supuestos realizamos al formular esta ecuación?
   - n es una potencia de 2 y que $(a_1 + a_2)$ y $(b_1 + b_2)$ tienen $\frac{n}{2}$ dígitos cada uno.
2. ¿Qué representa la constante e?
   - Calcular $(a_1 + a_2)$, $(b_1 + b_2)$, $(c_1 + c_2)$ y $(c_3 - (c_1 + c_2))$.
   - Construir $a \cdot b$ a partir de c_1, c_2 y $(c_3 - (c_1 + c_2))$, lo cual puede tomar tiempo lineal en el peor caso. ¿Por qué?

Resolviendo la ecuación de recurrencia

Utilizando el **Teorema Maestro** obtenemos que $T(n)$ es $\Theta(n^{\log_2(3)})$.

- Pero este resultado es válido bajo los supuestos realizados anteriormente.

¿Cómo debe formularse el algoritmo de Karatsuba en el **caso general**?

Caso general del algoritmo de Karatsuba

En el caso general, representamos las entradas a y b de la siguiente forma:

$$\begin{aligned} a &= a_1 \cdot 10^{\lfloor \frac{n}{2} \rfloor} + a_2 \\ b &= b_1 \cdot 10^{\lfloor \frac{n}{2} \rfloor} + b_2 \end{aligned}$$

La siguiente ecuación de recurrencia para $T(n)$ captura la cantidad de operaciones realizadas por el algoritmo (para constantes e_1, e_2):

$$T(n) = \begin{cases} e_1 & n \leq 3 \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lceil \frac{n}{2} \right\rceil + 1\right) + e_2 \cdot n & n > 3 \end{cases}$$

Ejercicio

Demuestre usando inducción constructiva que $T(n)$ es $\mathcal{O}(n^{\log_2(3)})$

- En particular, demuestre lo siguiente:
  $\exists c \in \mathbb{R}^+.\ \exists d \in \mathbb{R}^+.\ \exists n_0 \in \mathbb{N}.\ \forall n \geq n_0.\ T(n) \leq c \cdot n^{\log_2(3)} - d \cdot n$

Outline

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, ..., 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 $i \in \{1, \ldots, \ell - 1\}$

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

Ejemplo

$$\begin{tikzcd}
a \arrow[dr] & b \arrow[l] & e \arrow[l] \arrow[d] \\
 & d \arrow[ur] & f \arrow[l]
\end{tikzcd}$$

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 $V = \{1, \ldots, n\}$, $1 \leq \ell \leq n$ 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

$$
\begin{tikzcd}
1 \arrow[dr] & 2 \arrow[l] & 4 \arrow[l] \arrow[d] \\
 & 3 \arrow[ur] & 5 \arrow[l]
\end{tikzcd}
\qquad
M = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}
$$

Una primera definición de **ContarCaminos**

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

ContarCaminos($M[1 \ldots n][1 \ldots n]$, v_i, v_f, ℓ)
    if $\ell = 1$ **then return** $M[v_i, v_f]$
    else
        $aux := 0$
        for $v_j := 1$ **to** n **do**
            $aux \mathrel{+}= M[v_i, v_j] \cdot$ **ContarCaminos**(M, v_j, v_f, $\ell - 1$)
        return aux

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

¿Se puede mejorar este algoritmo?

Una segunda definición de **ContarCaminos**

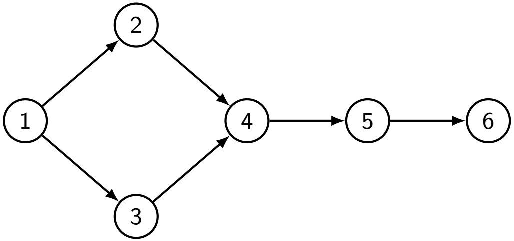
Podemos reducir el número de llamadas recursivas:

ContarCaminos$(M[1 \ldots n][1 \ldots n],\ v_i,\ v_f,\ \ell)$
    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**
                $aux \mathrel{+}=$ **ContarCaminos**$(M,\ v_j,\ v_f,\ \ell - 1)$
        return aux

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

Llamadas repetidas en **ContarCaminos**

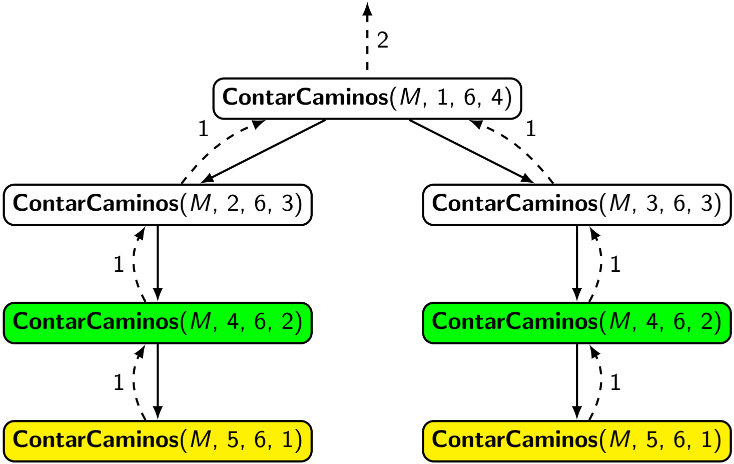
Considere el siguiente grafo G (representado por una matriz M):



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 $\dfrac{n^\ell - 1}{n-1}$ 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**

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** $H_1, \ldots, H_\ell$ 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 $H_1, \ldots, H_\ell$ se puede definir recursivamente de la siguiente forma:

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

¿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:

ContarTodosCaminos($M[1 \dots n][1 \dots n]$, ℓ)
    if $\ell = 1$ **then return** M
    else
        $H :=$ **ContarTodosCaminos**(M, $\ell - 1$)
        return $M \cdot H$

ContarCaminos($M[1 \dots n][1 \dots n]$, v_i, v_f, ℓ)
    $H :=$ **ContarTodosCaminos**(M, ℓ)
    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 mátrices para obtener este resultado

Outline

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

Sea $\Sigma = \{a, b, \ldots, z\}$ 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 $\mathrm{ed}(w_1, w_2)$ para la edit distance entre w_1 y w_2

- Tenemos que $\mathrm{ed} : \Sigma^* \times \Sigma^* \to \mathbb{N}$

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

$$\begin{aligned}
\text{eliminar}(w, i) &= a_1 \cdots a_{i-1} a_{i+1} \cdots a_n \\
\text{agregar}(w, i, b) &= a_1 \cdots a_i b a_{i+1} \cdots a_n \\
\text{cambiar}(w, i, b) &= a_1 \cdots a_{i-1} b a_{i+1} \cdots a_n
\end{aligned}$$

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

Ejemplo

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

La distancia de Levenshtein

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

Ejemplo

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

$$
\begin{aligned}
\text{agregar}(\texttt{casa}, 4, \texttt{d}) &= \texttt{casad} \\
\text{eliminar}(\texttt{casad}, 1) &= \texttt{asad} \\
\text{agregar}(\texttt{asad}, 4, \texttt{o}) &= \texttt{asado}
\end{aligned}
$$

La distancia de Levenshtein

Ejercicios

1. Demuestre que $\mathrm{ed}(w_1, w_2) \leq |w_1| + |w_2|$

2. Demuestre que ed es una función de distancia, vale decir, muestre lo siguiente:

   2.1 $\mathrm{ed}(w_1, w_2) = 0$ si y sólo si $w_1 = w_2$
   2.2 $\mathrm{ed}(w_1, w_2) = \mathrm{ed}(w_2, w_1)$
   2.3 $\mathrm{ed}(w_1, w_2) \leq \mathrm{ed}(w_1, w_3) + \mathrm{ed}(w_3, w_2)$

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 $\mathrm{hd}(w_1, w_2)$

   3.1 Demuestre que $\mathrm{ed}(w_1, w_2) \leq \mathrm{hd}(w_1, w_2)$
   3.2 Encuentre palabras w_3 y w_4 tales que $\mathrm{ed}(w_3, w_4) < \mathrm{hd}(w_3, w_4)$

Calculando la distancia de Levenshtein

Para calcular $\mathrm{ed}(w_1, w_2)$ 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 $\mathrm{ed}(w_1, w_2)$ utilizamos **programación dinámica**.

Notación preliminar

Dado $w \in \Sigma^*$ tal que $|w| = n$, y dado $i \in \{1, \ldots, n\}$, 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:

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

La distancia y un principio optimalidad para sub-secuencias

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

Suponga que o_1, ..., 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, ..., o_ℓ es la sub-secuencia de las operaciones en o_1, ..., 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: si para $s \in \{1, \ldots, \ell - 1\}$ 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 $o_{s+1}, o_s^\star$ tal que:

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

Finalmente, suponga que la aplicación de o_1, ..., 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 $o_1, \ldots, o_\ell$ 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 $o'_1, \ldots, o'_m$ es una secuencia de operaciones con $m < \ell$ que genera $w_2[1, j]$ desde $w_1[1, i]$

En la secuencia $o_1, \ldots, o_k$ que genera w_2 desde w_1 podemos reemplazar $o_1, \ldots, o_\ell$ por $o'_1, \ldots, o'_m$

- 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. $\square$

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

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

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

Además, definimos el valor $\mathrm{dif}(i, j)$ 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:

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

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$(w_1,\ i,\ w_2,\ j)$
    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 $d := 1$
        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):

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

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

		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	**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**

Estas operaciones son las siguientes:

$$\begin{aligned} \text{eliminar}(\texttt{casa}, 1) &= \texttt{asa} \\ \text{agregar}(\texttt{asa}, 3, \texttt{d}) &= \texttt{asad} \\ \text{agregar}(\texttt{asad}, 4, \texttt{o}) &= \texttt{asado} \end{aligned}$$

¿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.

Corolario

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