Técnicas fundamentales de diseño de algoritmos

IIC2283

Técnicas de diseño de algoritmos

En este capítulo vamos a estudiar tres técnicas fundamentales para el diseño de algoritmos:

- Dividir para conquistar
- Programación dinámica
- Algoritmos codiciosos

Estas tres técnicas son útiles para desarrollar algoritmos en muchos escenarios.

Dividir para conquistar

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

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$(S_1, \ldots, S_\ell)$

Dividir para conquistar

- En **DividirParaConquistar** k es un constante que indica cuando el tamaño de una entrada w es considerado pequeño: $|w| \leq k$

Dividir para conquistar

- En **DividirParaConquistar** k es un constante que indica cuando el tamaño de una entrada w es considerado pequeño: $|w| \leq k$

- Las entradas pequeñas son solucionadas utilizado un algoritmo diseñado para ellas: **InstanciasPequeñas**
  - En general este algoritmo es sencillo y ejecuta un número pequeño de operaciones

Dividir para conquistar

- En **DividirParaConquistar** k es un constante que indica cuando el tamaño de una entrada w es considerado pequeño: $|w| \leq k$
- Las entradas pequeñas son solucionadas utilizado un algoritmo diseñado para ellas: **InstanciasPequeñas**
  - En general este algoritmo es sencillo y ejecuta un número pequeño de operaciones
- Si el tamaño de una entrada w no es pequeño ($|w| > k$), entonces w es dividido es una secuencia $w_1, \ldots, w_\ell$ de entradas de menor tamaño para **DividirParaConquistar**

Dividir para conquistar

- Para cada $i \in \{1, \ldots, \ell\}$ la llamada **DividirParaConquistar**(w_i) resuelve el problema para la entrada w_i
  - El resultado de esta llamada es almacenado en S_i

Dividir para conquistar

- Para cada $i \in \{1, \ldots, \ell\}$ la llamada **DividirParaConquistar**(w_i) resuelve el problema para la entrada w_i
  - El resultado de esta llamada es almacenado en S_i
- Finalmente la llamada **Combinar**$(S_1, \ldots, S_\ell)$ combina los resultados $S_1, \ldots, S_\ell$ para obtener el resultados para w

Dividir para conquistar

- Para cada $i \in \{1, \ldots, \ell\}$ la llamada **DividirParaConquistar**(w_i) resuelve el problema para la entrada w_i
  - El resultado de esta llamada es almacenado en S_i
- Finalmente la llamada **Combinar**$(S_1, \ldots, S_\ell)$ combina los resultados $S_1, \ldots, S_\ell$ para obtener el resultados para w

Vimos un ejemplo de esta técnica en el algoritmo de búsqueda binaria

Dividir para conquistar

- Para cada $i \in \{1, \ldots, \ell\}$ la llamada **DividirParaConquistar**(w_i) resuelve el problema para la entrada w_i
  - El resultado de esta llamada es almacenado en S_i
- Finalmente la llamada **Combinar**$(S_1, \ldots, S_\ell)$ combina los resultados $S_1, \ldots, S_\ell$ para obtener el resultados para w

Vimos un ejemplo de esta técnica en el algoritmo de búsqueda binaria

- Vamos a ver como utilizar esta técnica para obtener un algoritmo eficiente para la multiplicación de dos números enteros

Antes de multiplicar: suma de números enteros

Sean a y b dos números enteros con $n \geq 1$ dígitos cada uno.

Primero queremos obtener $c = a + b$

Vamos a utilizar el algoritmo usual para calcular c

Antes de multiplicar: suma de números enteros

Sean a y b dos números enteros con $n \geq 1$ dígitos cada uno.

Primero queremos obtener $c = a + b$

Vamos a utilizar el algoritmo usual para calcular c

- Consideramos la suma de dos dígitos, la comparación de dos dígitos y la 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. ¿Tiene sentido suponer que todas tienen el mismo costo?

Antes de multiplicar: suma de números enteros

Sean a y b dos números enteros con $n \geq 1$ dígitos cada uno.

Primero queremos obtener $c = a + b$

Vamos a utilizar el algoritmo usual para calcular c

- Consideramos la suma de dos dígitos, la comparación de dos dígitos y la 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. ¿Tiene sentido suponer que todas tienen el mismo costo?
- ¿Cuál es el peor caso para el algoritmo usual? ¿Cuántas operaciones realiza el algoritmo en este caso?

Antes de multiplicar: suma de números enteros

Sean a y b dos números enteros con $n \geq 1$ dígitos cada uno.

Primero queremos obtener $c = a + b$

Vamos a utilizar el algoritmo usual para calcular c

- Consideramos la suma de dos dígitos, la comparación de dos dígitos y la 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. ¿Tiene sentido suponer que todas tienen el mismo costo?
- ¿Cuál es el peor caso para el algoritmo usual? ¿Cuántas operaciones realiza el algoritmo en este caso?
- ¿Cuántos dígitos puede tener c?

Multiplicación de números enteros

Ahora queremos obtener $d = a \cdot b$

Primero vamos a utilizar el algoritmo usual para calcular d

Multiplicación de números enteros

Ahora queremos obtener $d = a \cdot b$

Primero vamos a utilizar el algoritmo usual para calcular d

- Consideramos la suma y la multiplicación de dígitos como las operaciones a contar, ambas con costo 1. ¿Tienes sentido suponer que ambas operaciones tienen el mismo costo?

Multiplicación de números enteros

Ahora queremos obtener $d = a \cdot b$

Primero vamos a utilizar el algoritmo usual para calcular d

- Consideramos la suma y la multiplicación de dígitos como las operaciones a contar, ambas con costo 1. ¿Tienes sentido suponer que ambas operaciones tienen el mismo costo?
- ¿Cuál es el peor caso para el algoritmo usual? ¿Cuántas operaciones realiza el algoritmo en este caso?

Multiplicación de números enteros

Ahora queremos obtener $d = a \cdot b$

Primero vamos a utilizar el algoritmo usual para calcular d

- Consideramos la suma y la multiplicación de dígitos como las operaciones a contar, ambas con costo 1. ¿Tienes sentido suponer que ambas operaciones tienen el mismo costo?
- ¿Cuál es el peor caso para el algoritmo usual? ¿Cuántas operaciones realiza el algoritmo en este caso?
- ¿Cuántos dígitos puede tener d?

Dividir para conquistar: el algoritmo de multiplicación de Karatsuba

Suponga que a y b son dos números con n dígitos cada uno, donde n es una potencia de 2.

Dividir para conquistar: el algoritmo de multiplicación de Karatsuba

Suponga que a y b son dos números con n dígitos cada uno, donde n es una potencia de 2.

Podemos representar a y b de la siguiente forma:

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

Dividir para conquistar: el algoritmo de multiplicación de Karatsuba

Suponga que a y b son dos números con n dígitos cada uno, donde n es una potencia de 2.

Podemos representar a y b de la siguiente forma:

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

Tenemos entonces que:

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

El algoritmo de multiplicación de Karatsuba

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. $a_2 \cdot b_2$

El algoritmo de multiplicación de Karatsuba

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 $\frac{n}{2}$

El algoritmo de multiplicación de Karatsuba

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?

El algoritmo de multiplicación de Karatsuba

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?

- Puede usar el teorema Maestro para deducir que este algoritmo es de orden $\Theta(n^2)$

La idea clave en el algoritmo de Karatsuba

Para calcular $a \cdot b$ realizamos las siguientes multiplicaciones:

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

La idea clave en el algoritmo de Karatsuba

Para calcular $a \cdot b$ realizamos 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:

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

La idea clave en el algoritmo de Karatsuba

Para calcular $a \cdot b$ realizamos 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:

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

¿Cuántas operaciones realiza este algoritmo?

El tiempo de ejecución del algoritmo de Karatsuba

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

El tiempo de ejecución del algoritmo de Karatsuba

$T(n)$: 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 una constante):

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

El tiempo de ejecución del algoritmo de Karatsuba

¿Qué supuestos realizamos al formular esta ecuación?

El tiempo de ejecución del algoritmo de Karatsuba

¿Qué supuestos realizamos al formular esta ecuación?

- n es una potencia de 2
- $(a_1 + a_2)$ y $(b_1 + b_2)$ tienes $\frac{n}{2}$ dígitos cada uno

El tiempo de ejecución del algoritmo de Karatsuba

¿Qué supuestos realizamos al formular esta ecuación?

- n es una potencia de 2
- $(a_1 + a_2)$ y $(b_1 + b_2)$ tienes $\frac{n}{2}$ dígitos cada uno

¿Qué representa la constante e?

El tiempo de ejecución del algoritmo de Karatsuba

¿Qué supuestos realizamos al formular esta ecuación?

- n es una potencia de 2
- $(a_1 + a_2)$ y $(b_1 + b_2)$ tienes $\frac{n}{2}$ dígitos cada uno

¿Qué representa la constante e? Utilizamos e para codificar el números de operaciones necesarias para:

- 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 $\Theta(n^{\log_2(3)})$

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

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?

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?¿Cuál es la complejidad de este algoritmo?

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?¿Cuál es la complejidad de este algoritmo?

- La clave para realizar el análisis del algoritmo de Karatsuba es el uso de ecuaciones de recurrencia e indución constructiva

La complejidad 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 complejidad 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):

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

La complejidad 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):

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

Ejercicio

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

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.

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

Dado un grafo $G = (N, A)$, recuerde que una secuencia $a_1, \ldots, a_\ell$ de elementos en N es un camino en G si:

1. $\ell \geq 2$
2. $(a_i, a_{i+1}) \in A$ para cada $i \in \{1, \ldots, \ell - 1\}$

Decimos que un camino $a_1, \ldots, a_\ell$ va desde a_1 a a_ℓ, y definimos su largo como (ℓ - 1), vale decir, el número de arcos en el camino.

Contando el número de caminos en un grafo

Dado un grafo $G = (N, A)$, un par de nodos b, c en N y un número ℓ, queremos desarrollar un algoritmo que cuente el número de caminos desde b a c en G cuyo largo es igual a ℓ

Contando el número de caminos en un grafo

Dado un grafo $G = (N, A)$, un par de nodos b, c en N y un número ℓ, queremos desarrollar un algoritmo que cuente el número de caminos desde b a c en G cuyo largo es igual a ℓ

Suponemos que $N = \{1, \ldots, n\}$, $1 \leq \ell \leq n$ y representamos G a través de su matriz de adyacencia M:

- si $(i, j) \in A$, entonces $M[i, j] = 1$, en caso contrario $M[i, j] = 0$

Contando el número de caminos en un grafo

Dado un grafo $G = (N, A)$, un par de nodos b, c en N y un número ℓ, queremos desarrollar un algoritmo que cuente el número de caminos desde b a c en G cuyo largo es igual a ℓ

Suponemos que $N = \{1, \ldots, n\}$, $1 \leq \ell \leq n$ y representamos G a través de su matriz de adyacencia M:

- si $(i, j) \in A$, entonces $M[i, j] = 1$, en caso contrario $M[i, j] = 0$

Queremos entonces definir la función **ContarCaminos**(M, b, c, ℓ)

Una primera definición de **ContarCaminos**

ContarCaminos$(M[1 \ldots n][1 \ldots n],\ b,\ c,\ \ell)$
    if $\ell = 1$ **then return** $M[b, c]$
    else
        $aux := 0$
        for $i := 1$ **to** n **do**
            $aux := aux + M[b, i] \cdot$ **ContarCaminos**$(M,\ i,\ c,\ \ell - 1)$
        return aux

Una primera definición de **ContarCaminos**

ContarCaminos$(M[1 \ldots n][1 \ldots n],\ b,\ c,\ \ell)$
    if $\ell = 1$ **then return** $M[b, c]$
    else
        $aux := 0$
        for $i := 1$ **to** n **do**
            $aux := aux + M[b, i] \cdot$ **ContarCaminos**$(M,\ i,\ c,\ \ell - 1)$
        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.

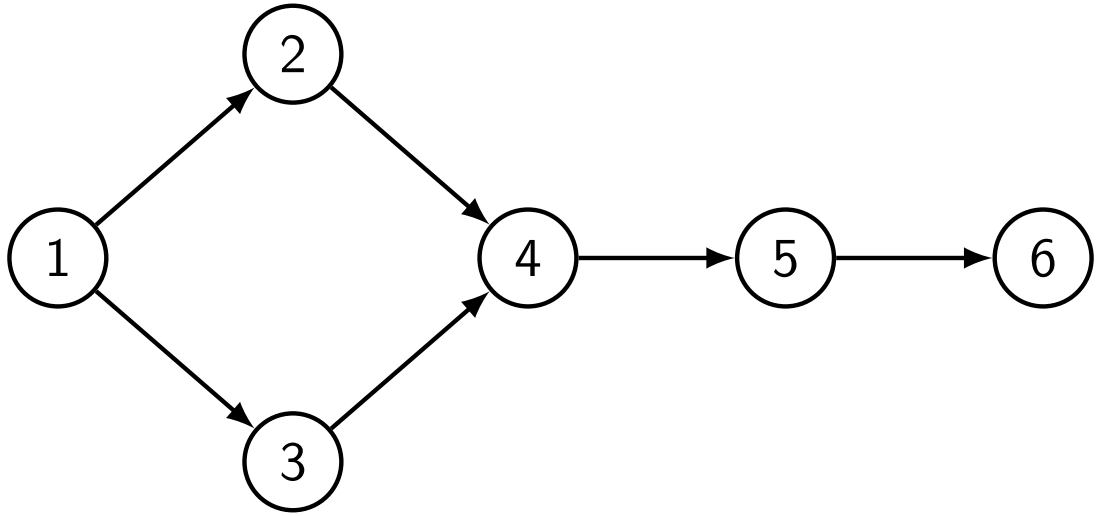
Una segunda definición de **ContarCaminos**

Podemos reducir el número de llamadas recursivas:

ContarCaminos$(M[1 \ldots n][1 \ldots n],\ b,\ c,\ \ell)$
    if $\ell = 1$ **then return** $M[b, c]$
    else
        $aux := 0$
        for $i := 1$ **to** n **do**
            if $M[b, i] = 1$ **then**
                $aux := aux +$ **ContarCaminos**$(M,\ i,\ c,\ \ell - 1)$
        return aux

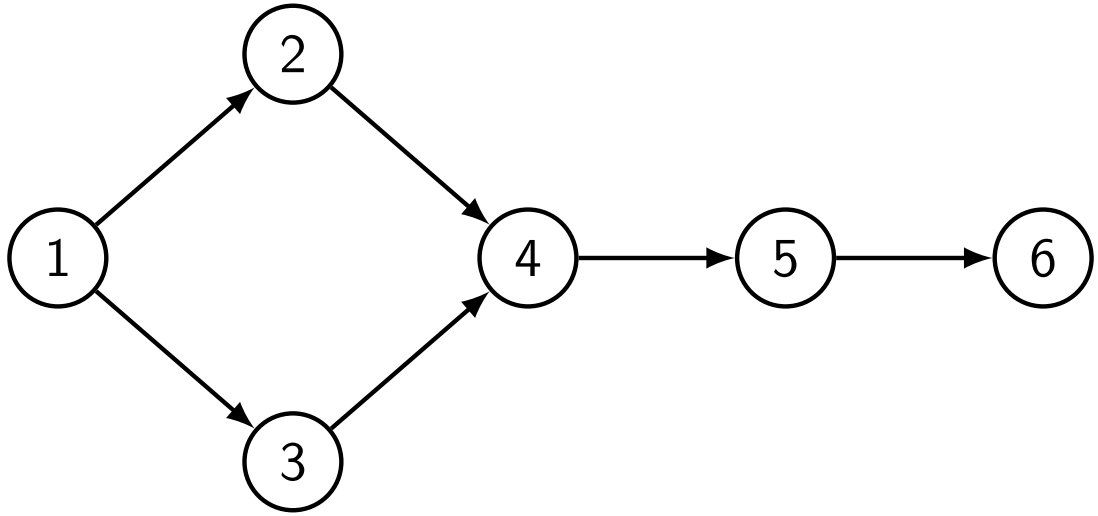
Llamadas repetidas en **ContarCaminos**

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



Llamadas repetidas en **ContarCaminos**

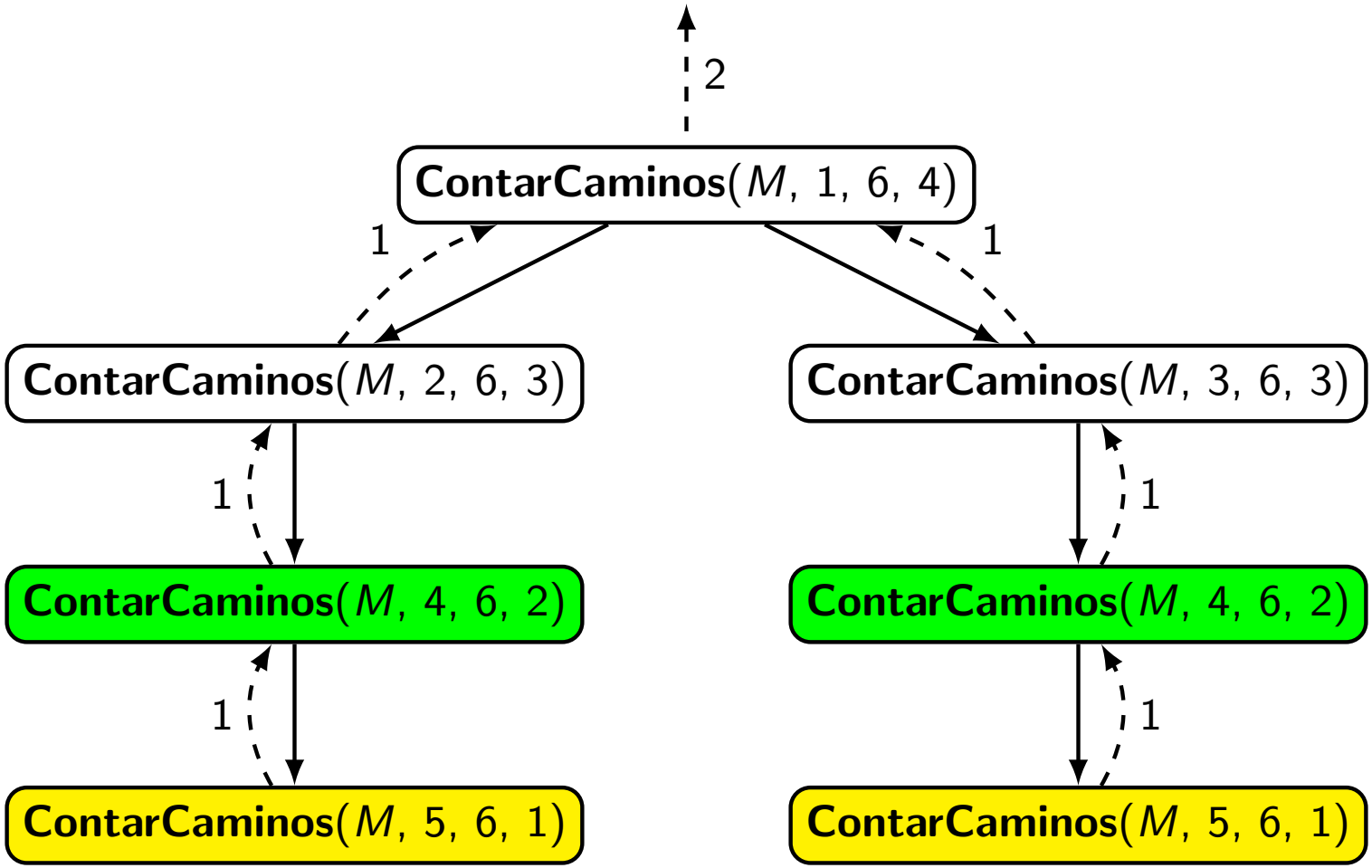
Considere el siguiente grafo G (representado por una matriz M):



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 debe realizar $\frac{n^\ell - 1}{n - 1}$ llamadas al procedimiento **ContarCaminos**, suponiendo que la entrada es $M[1 \ldots n][1 \ldots n]$, b, c, ℓ con $n \geq 2$

- ¿Para qué grafos obtenemos el peor caso?

Llamadas repetidas en **ContarCaminos**

Ejercicio

Demuestre que en el peor caso se debe realizar $\frac{n^\ell - 1}{n - 1}$ llamadas al procedimiento **ContarCaminos**, suponiendo que la entrada es $M[1 \ldots n][1 \ldots n]$, b, c, ℓ con $n \geq 2$

- ¿Para qué grafos obtenemos el peor caso?

El algoritmo realiza muchas llamadas recursivas repetidas.

Llamadas repetidas en **ContarCaminos**

Ejercicio

Demuestre que en el peor caso se debe realizar $\frac{n^\ell - 1}{n - 1}$ llamadas al procedimiento **ContarCaminos**, suponiendo que la entrada es $M[1 \ldots n][1 \ldots n]$, b, c, ℓ con $n \geq 2$

- ¿Para qué grafos obtenemos el peor caso?

El algoritmo realiza muchas llamadas recursivas repetidas.

- Un número exponencial dado que sólo podemos tener $n^2 \cdot \ell$ llamadas distintas en la ejecución de **ContarCaminos**$(M[1 \ldots n][1 \ldots n], b, c, \ell)$

Llamadas repetidas en **ContarCaminos**

Ejercicio

Demuestre que en el peor caso se debe realizar $\frac{n^\ell - 1}{n - 1}$ llamadas al procedimiento **ContarCaminos**, suponiendo que la entrada es $M[1 \ldots n][1 \ldots n]$, b, c, ℓ con $n \geq 2$

- ¿Para qué grafos obtenemos el peor caso?

El algoritmo realiza muchas llamadas recursivas repetidas.

- Un número exponencial dado que sólo podemos tener $n^2 \cdot \ell$ llamadas distintas en la ejecución de **ContarCaminos**$(M[1 \ldots n][1 \ldots n], b, c, \ell)$

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 b a un nodo c 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 b a un nodo c 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:

$$H_k[i,j] \text{ es el número de caminos de } i \text{ a } j \text{ de largo } k$$

De esta forma la respuesta a la pregunta inicial está en $H_\ell[b,c]$

Una tercera definición de **ContarCaminos**

Queremos calcular el número de caminos de largo ℓ desde un nodo b a un nodo c 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:

$H_k[i,j]$ es el número de caminos de i a j de largo k

De esta forma la respuesta a la pregunta inicial está en $H_\ell[b,c]$

La secuencia $H_1, \ldots, H_\ell$ se define de la siguiente forma:

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

Una tercera definición de **ContarCaminos**

La implementación recursiva de la idea descrita:

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

ContarCaminos$(M[1 \ldots n][1 \ldots n], b, c, \ell)$
    $H :=$ **ContarTodosCaminos**(M, ℓ)
    return $H[b, c]$

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

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:

> 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

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

- Vamos a considerar las palabras (strings) sobre el alfabeto Σ

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 (strings) sobre el alfabeto Σ

Vamos a utilizar 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 $\Sigma = \{a, b, \ldots, z\}$ el alfabeto español, el cual contiene 27 símbolos.

- Vamos a considerar las palabras (strings) sobre el alfabeto Σ

Vamos a utilizar 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 $\text{ed} : \Sigma^* \times \Sigma^* \to \mathbb{N}$

Tres operadores sobre palabras

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

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

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 $\text{agregar}(w, 0, b) = bw$

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

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

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 $\text{ed}(w_1, w_2) \leq |w_1| + |w_2|$

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 $\text{ed}(w_1, w_2)$ 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 $\text{ed}(w_1, w_2)$ 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 $\text{ed}(w_1, w_2)$ 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 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 $w_1, w_2 \in \Sigma^*$, y suponga que $o_1, \ldots, o_k$ 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 $w_1, w_2 \in \Sigma^*$, y suponga que $o_1, \ldots, o_k$ 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 $o_1, \ldots, o_\ell$ es la sub-secuencia de las operaciones en $o_1, \ldots, o_k$ que son aplicadas a $w_1[1, i]$ con $1 \leq \ell \leq k$

La distancia y un principio optimalidad para sub-secuencias

Sean $w_1, w_2 \in \Sigma^*$, y suponga que $o_1, \ldots, o_k$ 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 $o_1, \ldots, o_\ell$ es la sub-secuencia de las operaciones en $o_1, \ldots, o_k$ 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 $t - 1$

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 $o_1, \ldots, o_\ell$ 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, $\text{ed}(w_1[1, i], w_2[1, j]) = \ell$

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

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 $w_1[1, i]$

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

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

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

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.

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

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

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:

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

Una definición recursiva de la distancia de Levenshtein

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

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

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

Una definición recursiva de la distancia de Levenshtein

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

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

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 $\text{mín}\{1 + r, 1 + s, d + t\}$

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
        $r :=$ **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 $\text{mín}\{1+r, 1+s, d+t\}$

¿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
    $r :=$ **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 $\text{mín}\{1+r, 1+s, d+t\}$

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

- 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):

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

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(casa, 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**

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:

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

El costo de calcular la distancia de Levenshtein

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

El costo de calcular la distancia de Levenshtein

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

- 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

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

- 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* $\Theta(|w_1| \cdot |w_2|)$

Algoritmos codiciosos

Los algoritmos codiciosos son usados, en general, para resolver problemas de optimización.

Algoritmos codiciosos

Los algoritmos codiciosos son usados, en general, para resolver problemas de optimización.

El principio fundamental de este tipo de algoritmos es lograr un óptimo global tomando decisiones locales que son óptimas.

Algoritmos codiciosos

Los algoritmos codiciosos son usados, en general, para resolver problemas de optimización.

El principio fundamental de este tipo de algoritmos es lograr un óptimo global tomando decisiones locales que son óptimas.

- En general, la intuición detrás de estos algoritmos es simple

Algoritmos codiciosos

Los algoritmos codiciosos son usados, en general, para resolver problemas de optimización.

El principio fundamental de este tipo de algoritmos es lograr un óptimo global tomando decisiones locales que son óptimas.

- En general, la intuición detrás de estos algoritmos es simple

Dos componentes fundamentales de un algoritmo codicioso:

- Una función objetivo a minimizar o maximizar
- Una función de selección usada para tomar decisiones locales óptimas

Algoritmos codiciosos

Lamentablemente en muchos casos los algoritmos codiciosos no encuentran un óptimo global

Algoritmos codiciosos

Lamentablemente en muchos casos los algoritmos codiciosos no encuentran un óptimo global

- Sin embargo, en muchos casos pueden ser usados como heurísticas para encontrar valores de la función objetivo cercanos al óptimo

Algoritmos codiciosos

Lamentablemente en muchos casos los algoritmos codiciosos no encuentran un óptimo global

- Sin embargo, en muchos casos pueden ser usados como heurísticas para encontrar valores de la función objetivo cercanos al óptimo

Una segunda dificultad con los algoritmos codiciosos es que, en general, se necesita de una demostración formal para asegurar que encuentran el óptimo global

Algoritmos codiciosos

Lamentablemente en muchos casos los algoritmos codiciosos no encuentran un óptimo global

- Sin embargo, en muchos casos pueden ser usados como heurísticas para encontrar valores de la función objetivo cercanos al óptimo

Una segunda dificultad con los algoritmos codiciosos es que, en general, se necesita de una demostración formal para asegurar que encuentran el óptimo global

- Aunque pueden ser algoritmos simples, en algunos casos no es obvio por qué encuentran el óptimo global

Algoritmos codiciosos

Lamentablemente en muchos casos los algoritmos codiciosos no encuentran un óptimo global

- Sin embargo, en muchos casos pueden ser usados como heurísticas para encontrar valores de la función objetivo cercanos al óptimo

Una segunda dificultad con los algoritmos codiciosos es que, en general, se necesita de una demostración formal para asegurar que encuentran el óptimo global

- Aunque pueden ser algoritmos simples, en algunos casos no es obvio por qué encuentran el óptimo global

Vamos a ver un ejemplo clásico de algoritmos codiciosos.

Almacenamiento de datos

Dada una palabra w sobre un alfabeto Σ, suponga que queremos almacenar w utilizando los símbolos 0 y 1

Almacenamiento de datos

Dada una palabra w sobre un alfabeto Σ, suponga que queremos almacenar w utilizando los símbolos 0 y 1

- Esta palabra puede ser pensada como un texto si Σ incluye las letras del alfabeto español, símbolos de puntuación y el espacio, por lo tanto puede ser muy larga
- El uso de 0 y 1 corresponde a la idea de almacenar el texto en un computador

Almacenamiento de datos

Dada una palabra w sobre un alfabeto Σ, suponga que queremos almacenar w utilizando los símbolos 0 y 1

- Esta palabra puede ser pensada como un texto si Σ incluye las letras del alfabeto español, símbolos de puntuación y el espacio, por lo tanto puede ser muy larga
- El uso de 0 y 1 corresponde a la idea de almacenar el texto en un computador

Definimos entonces una función $\tau : \Sigma \to \{0,1\}^*$ que asigna a cada símbolo en $a \in \Sigma$ una palabra en $\tau(a) \in \{0,1\}^*$ con $\tau(a) \neq \varepsilon$

Almacenamiento de datos

Dada una palabra w sobre un alfabeto Σ, suponga que queremos almacenar w utilizando los símbolos 0 y 1

- Esta palabra puede ser pensada como un texto si Σ incluye las letras del alfabeto español, símbolos de puntuación y el espacio, por lo tanto puede ser muy larga
- El uso de 0 y 1 corresponde a la idea de almacenar el texto en un computador

Definimos entonces una función $\tau : \Sigma \to \{0,1\}^*$ que asigna a cada símbolo en $a \in \Sigma$ una palabra en $\tau(a) \in \{0,1\}^*$ con $\tau(a) \neq \varepsilon$

- Vamos a almacenar w reemplazando cada símbolo $a \in \Sigma$ que aparece en w por $\tau(a)$

Almacenamiento de datos

Dada una palabra w sobre un alfabeto Σ, suponga que queremos almacenar w utilizando los símbolos 0 y 1

- Esta palabra puede ser pensada como un texto si Σ incluye las letras del alfabeto español, símbolos de puntuación y el espacio, por lo tanto puede ser muy larga
- El uso de 0 y 1 corresponde a la idea de almacenar el texto en un computador

Definimos entonces una función $\tau : \Sigma \to \{0,1\}^*$ que asigna a cada símbolo en $a \in \Sigma$ una palabra en $\tau(a) \in \{0,1\}^*$ con $\tau(a) \neq \varepsilon$

- Vamos a almacenar w reemplazando cada símbolo $a \in \Sigma$ que aparece en w por $\tau(a)$
- Llamamos a τ una Σ-codificación

Almacenamiento de datos

La extensión $\hat{\tau}$ de una Σ-codificación τ a todas las palabras $w \in \Sigma^*$ se define como:

$$\hat{\tau}(w) \quad = \quad \begin{cases} \varepsilon & w = \varepsilon \\ \tau(a_1) \cdots \tau(a_n) & w = a_1 \cdots a_n \text{ con } n \geq 1 \end{cases}$$

Almacenamiento de datos

La extensión $\hat{\tau}$ de una Σ-codificación τ a todas las palabras $w \in \Sigma^*$ se define como:

$$\hat{\tau}(w) \quad = \quad \begin{cases} \varepsilon & w = \varepsilon \\ \tau(a_1) \cdots \tau(a_n) & w = a_1 \cdots a_n \text{ con } n \geq 1 \end{cases}$$

Vamos a almacenar w como $\hat{\tau}(w)$

Almacenamiento de datos

La extensión $\hat{\tau}$ de una Σ-codificación τ a todas las palabras $w \in \Sigma^*$ se define como:

$$\hat{\tau}(w) = \begin{cases} \varepsilon & w = \varepsilon \\ \tau(a_1) \cdots \tau(a_n) & w = a_1 \cdots a_n \text{ con } n \geq 1 \end{cases}$$

Vamos a almacenar w como $\hat{\tau}(w)$

- Si la Σ-codificación τ está fija (como el código ASCII) entonces no es necesario almacenarla

Almacenamiento de datos

La extensión $\hat{\tau}$ de una Σ-codificación τ a todas las palabras $w \in \Sigma^*$ se define como:

$$\hat{\tau}(w) = \begin{cases} \varepsilon & w = \varepsilon \\ \tau(a_1) \cdots \tau(a_n) & w = a_1 \cdots a_n \text{ con } n \geq 1 \end{cases}$$

Vamos a almacenar w como $\hat{\tau}(w)$

- Si la Σ-codificación τ está fija (como el código ASCII) entonces no es necesario almacenarla
- Si τ no está fija entonces debemos almacenarla junto con $\hat{\tau}(w)$
  - En general $|w|$ es mucho más grande que $|\Sigma|$, por lo que el costo de almacenar τ es despreciable

Almacenamiento de datos

La función $\hat{\tau}$ debe especificar una traducción no ambigua: si $w_1 \neq w_2$, entonces $\hat{\tau}(w_1) \neq \hat{\tau}(w_2)$

- De esta forma podemos reconstruir el texto original dada su traducción

Almacenamiento de datos

La función $\hat{\tau}$ debe especificar una traducción no ambigua: si $w_1 \neq w_2$, entonces $\hat{\tau}(w_1) \neq \hat{\tau}(w_2)$

- De esta forma podemos reconstruir el texto original dada su traducción

Vale decir, $\hat{\tau}$ debe ser una función inyectiva

Almacenamiento de datos

La función $\hat{\tau}$ debe especificar una traducción no ambigua: si $w_1 \neq w_2$, entonces $\hat{\tau}(w_1) \neq \hat{\tau}(w_2)$

- De esta forma podemos reconstruir el texto original dada su traducción

Vale decir, $\hat{\tau}$ debe ser una función inyectiva

- ¿Cómo podemos lograr esta propiedad?

Codificaciones de largo fijo

Obviamente para lograr una traducción no ambigua necesitamos que $\tau(a) \neq \tau(b)$ para cada $a, b \in \Sigma$ tal que $a \neq b$

Codificaciones de largo fijo

Obviamente para lograr una traducción no ambigua necesitamos que $\tau(a) \neq \tau(b)$ para cada $a, b \in \Sigma$ tal que $a \neq b$

Para obtener la misma propiedad para $\hat{\tau}$ podemos imponer la siguiente condición: para cada $a, b \in \Sigma$ se tiene que $|\tau(a)| = |\tau(b)|$

Codificaciones de largo fijo

Obviamente para lograr una traducción no ambigua necesitamos que $\tau(a) \neq \tau(b)$ para cada $a, b \in \Sigma$ tal que $a \neq b$

Para obtener la misma propiedad para $\hat{\tau}$ podemos imponer la siguiente condición: para cada $a, b \in \Sigma$ se tiene que $|\tau(a)| = |\tau(b)|$

- Decimos entonces que τ es una Σ-codificación de largo fijo

Codificaciones de largo fijo

Obviamente para lograr una traducción no ambigua necesitamos que $\tau(a) \neq \tau(b)$ para cada $a, b \in \Sigma$ tal que $a \neq b$

Para obtener la misma propiedad para $\hat{\tau}$ podemos imponer la siguiente condición: para cada $a, b \in \Sigma$ se tiene que $|\tau(a)| = |\tau(b)|$

- Decimos entonces que τ es una Σ-codificación de largo fijo

Ejercicio

Muestre que para tener una Σ-codificación de largo fijo basta con asignar a cada $a \in \Sigma$ una palabra $\tau(a) \in \{0,1\}^*$ tal que $|\tau(a)| = \lceil \log_2(|\Sigma|) \rceil$

Codificaciones de largo variable

Decimos que τ es una Σ-codificación de largo variable si existen $a, b \in \Sigma$ tales que $|\tau(a)| \neq |\tau(b)|$

Codificaciones de largo variable

Decimos que τ es una Σ-codificación de largo variable si existen $a, b \in \Sigma$ tales que $|\tau(a)| \neq |\tau(b)|$

¿Por qué nos conviene utilizar codificaciones de largo variable?

Codificaciones de largo variable

Decimos que τ es una Σ-codificación de largo variable si existen $a, b \in \Sigma$ tales que $|\tau(a)| \neq |\tau(b)|$

¿Por qué nos conviene utilizar codificaciones de largo variable?

- Podemos obtener representaciones más cortas para la palabra que queremos almacenar

Codificaciones de largo variable

Ejemplo

Suponga que $w = aabaacaab$

- Para $\tau_1(a) = 00$, $\tau_1(b) = 01$ y $\tau_1(c) = 10$ tenemos que:

$$\hat{\tau}_1(w) \quad = \quad 000001000010000001$$

- Para $\tau_2(a) = 0$, $\tau_2(b) = 10$ y $\tau_2(c) = 11$ tenemos que:

$$\hat{\tau}_2(w) \quad = \quad 001000110010$$

Por lo tanto $|\hat{\tau}_2(w)| = 12 < 18 = |\hat{\tau}_1(w)|$

Codificaciones de largo variable

Ejemplo

Suponga que $w = aabaacaab$

- Para $\tau_1(a) = 00$, $\tau_1(b) = 01$ y $\tau_1(c) = 10$ tenemos que:

$$\hat{\tau}_1(w) \quad = \quad 000001000010000001$$

- Para $\tau_2(a) = 0$, $\tau_2(b) = 10$ y $\tau_2(c) = 11$ tenemos que:

$$\hat{\tau}_2(w) \quad = \quad 001000110010$$

Por lo tanto $|\hat{\tau}_2(w)| = 12 < 18 = |\hat{\tau}_1(w)|$

Pero nos falta responder una pregunta: ¿por qué $\hat{\tau}_2$ es inyectiva?

Codificaciones de largo variable

Lema

Si existen $w_1, w_2 \in \Sigma^*$ *tales que* $w_1 \neq w_2$ *y* $\hat{\tau}(w_1) = \hat{\tau}(w_2)$*, entonces existen* $a, b \in \Sigma$ *tales que* $a \neq b$ *y* $\tau(a)$ *es un prefijo de* $\tau(b)$

Codificaciones de largo variable

Lema

Si existen $w_1, w_2 \in \Sigma^*$ *tales que* $w_1 \neq w_2$ *y* $\hat{\tau}(w_1) = \hat{\tau}(w_2)$*, entonces existen* $a, b \in \Sigma$ *tales que* $a \neq b$ *y* $\tau(a)$ *es un prefijo de* $\tau(b)$

Demostración: Suponga que $w_1 \neq w_2$, $\hat{\tau}(w_1) = \hat{\tau}(w_2)$ y

$$\begin{array}{lcll} w_1 & = & a_1 \dots a_m & m \geq 1 \\ w_2 & = & b_1 \dots b_n & n \geq 1 \end{array}$$

Además, sin perdida de generalidad suponga que $m \leq n$

Si w_1 es prefijo propio de w_2 entonces $\hat{\tau}(w_1)$ es prefijo propio de $\hat{\tau}(w_2)$

- Puesto que $\tau(a) \neq \varepsilon$ para cada $a \in \Sigma$

Dado que $\hat{\tau}(w_1) = \hat{\tau}(w_2)$, tenemos entonces que w_1 no es prefijo propio de w_2

Codificaciones de largo variable

Sea $k = \min_{1 \leq i \leq m} a_i \neq b_i$

- k está bien definido puesto que $w_1 \neq w_2$ y w_1 no es prefijo propio de w_2

Dado que $\hat{\tau}(a_1 \cdots a_{k-1}) = \hat{\tau}(b_1 \cdots b_{k-1})$ y $\hat{\tau}(w_1) = \hat{\tau}(w_2)$, concluimos que $\hat{\tau}(a_k \cdots a_m) = \hat{\tau}(b_k \cdots b_n)$

Tenemos entonces que $\tau(a_k) \cdots \tau(a_m) = \tau(b_k) \cdots \tau(b_n)$, de lo cual se deduce que $\tau(a_k)$ es un prefijo de $\tau(b_k)$ o $\tau(b_k)$ es un prefijo de $\tau(a_k)$

- Lo cual concluye la demostración puesto que $a_k \neq b_k$ □

Codificaciones libres de prefijos

Decimos que una Σ-codificación τ es libre de prefijos si para cada $a, b \in \Sigma$ tales que $a \neq b$ se tiene que $\tau(a)$ no es un prefijo de $\tau(b)$

Codificaciones libres de prefijos

Decimos que una Σ-codificación τ es libre de prefijos si para cada $a, b \in \Sigma$ tales que $a \neq b$ se tiene que $\tau(a)$ no es un prefijo de $\tau(b)$

Ejemplo

Para $\Sigma = \{a, b, c\}$ y $\tau(a) = 0$, $\tau(b) = 10$, $\tau(c) = 11$, se tiene que τ es libre de prefijos.

Codificaciones libres de prefijos

Decimos que una Σ-codificación τ es libre de prefijos si para cada $a, b \in \Sigma$ tales que $a \neq b$ se tiene que $\tau(a)$ no es un prefijo de $\tau(b)$

Ejemplo

Para $\Sigma = \{a, b, c\}$ y $\tau(a) = 0$, $\tau(b) = 10$, $\tau(c) = 11$, se tiene que τ es libre de prefijos.

Corolario

Si τ es una codificación libre de prefijos, entonces $\hat{\tau}$ es una función inyectiva.

Frecuencias relativas de los símbolos

Palabra a almacenar: $w \in \Sigma^*$

Frecuencias relativas de los símbolos

Palabra a almacenar: $w \in \Sigma^*$

Para $a \in \Sigma$ definimos $\text{fr}_w(a)$ como la frecuencia relativa de a en w, vale decir, el número de apariciones de a en w dividido por el largo de w

- Por ejemplo, si $w = aabaacaab$, entonces $\text{fr}_w(a) = \frac{2}{3}$ y $\text{fr}_w(c) = \frac{1}{9}$

Frecuencias relativas de los símbolos

Palabra a almacenar: $w \in \Sigma^*$

Para $a \in \Sigma$ definimos $\text{fr}_w(a)$ como la frecuencia relativa de a en w, vale decir, el número de apariciones de a en w dividido por el largo de w

- Por ejemplo, si $w = aabaacaab$, entonces $\text{fr}_w(a) = \frac{2}{3}$ y $\text{fr}_w(c) = \frac{1}{9}$

Para una Σ-codificación τ definimos el largo promedio para w como:

$$\text{lp}_w(\tau) \quad = \quad \sum_{a \in \Sigma} \text{fr}_w(a) \cdot |\tau(a)|$$

Frecuencias relativas de los símbolos

Palabra a almacenar: $w \in \Sigma^*$

Para $a \in \Sigma$ definimos $\mathrm{fr}_w(a)$ como la frecuencia relativa de a en w, vale decir, el número de apariciones de a en w dividido por el largo de w

- Por ejemplo, si $w = aabaacaab$, entonces $\mathrm{fr}_w(a) = \frac{2}{3}$ y $\mathrm{fr}_w(c) = \frac{1}{9}$

Para una Σ-codificación τ definimos el largo promedio para w como:

$$\mathrm{lp}_w(\tau) \quad = \quad \sum_{a \in \Sigma} \mathrm{fr}_w(a) \cdot |\tau(a)|$$

Tenemos entonces que $|\hat{\tau}(w)| = \mathrm{lp}_w(\tau) \cdot |w|$

Frecuencias relativas de los símbolos

Palabra a almacenar: $w \in \Sigma^*$

Para $a \in \Sigma$ definimos $\text{fr}_w(a)$ como la frecuencia relativa de a en w, vale decir, el número de apariciones de a en w dividido por el largo de w

- Por ejemplo, si $w = aabaacaab$, entonces $\text{fr}_w(a) = \frac{2}{3}$ y $\text{fr}_w(c) = \frac{1}{9}$

Para una Σ-codificación τ definimos el largo promedio para w como:

$$\text{lp}_w(\tau) \quad = \quad \sum_{a \in \Sigma} \text{fr}_w(a) \cdot |\tau(a)|$$

Tenemos entonces que $|\hat{\tau}(w)| = \text{lp}_w(\tau) \cdot |w|$

- Por lo tanto queremos una Σ-codificación τ que minimice $\text{lp}_w(\tau)$

Frecuencias relativas de los símbolos y la función objetivo

Problema de optimización a resolver

Dado $w \in \Sigma^*$, encontrar una Σ-codificación τ libre de prefijos que minimice el valor $\mathrm{lp}_w(\tau)$

Frecuencias relativas de los símbolos y la función objetivo

Problema de optimización a resolver

Dado $w \in \Sigma^*$, encontrar una Σ-codificación τ libre de prefijos que minimice el valor $\mathrm{lp}_w(\tau)$

La función objetivo del algoritmo codicioso es entonces $\mathrm{lp}_w(x)$

- Queremos minimizar el valor de esta función

Frecuencias relativas de los símbolos y la función objetivo

La función $\mathrm{lp}_w(x)$ se define a partir de la función $\mathrm{fr}_w(y)$

Frecuencias relativas de los símbolos y la función objetivo

La función $\mathrm{lp}_w(x)$ se define a partir de la función $\mathrm{fr}_w(y)$

- No se necesita más información sobre w. En particular, no se necesita saber cuál es el símbolo de w en una posición específica

Frecuencias relativas de los símbolos y la función objetivo

La función $\mathrm{lp}_w(x)$ se define a partir de la función $\mathrm{fr}_w(y)$

- No se necesita más información sobre w. En particular, no se necesita saber cuál es el símbolo de w en una posición específica

Podemos entonces trabajar con funciones de frecuencias relativas en lugar de palabras

- La entrada del problema no va a ser w sino que fr_w

Frecuencias relativas de los símbolos y la función objetivo

Decimos que $f : \Sigma \to (0, 1)$ es una función de frecuencias relativas para Σ si se cumple que $\sum_{a \in \Sigma} f(a) = 1$

Frecuencias relativas de los símbolos y la función objetivo

Decimos que $f : \Sigma \to (0, 1)$ es una función de frecuencias relativas para Σ si se cumple que $\sum_{a \in \Sigma} f(a) = 1$

Dada una función f de frecuencias relativas para Σ y una Σ-codificación τ, el largo promedio de τ para f se define como:

$$\mathrm{lp}_f(\tau) \quad = \quad \sum_{a \in \Sigma} f(a) \cdot |\tau(a)|$$

Frecuencias relativas de los símbolos y la función objetivo

Decimos que $f : \Sigma \to (0, 1)$ es una función de frecuencias relativas para Σ si se cumple que $\sum_{a \in \Sigma} f(a) = 1$

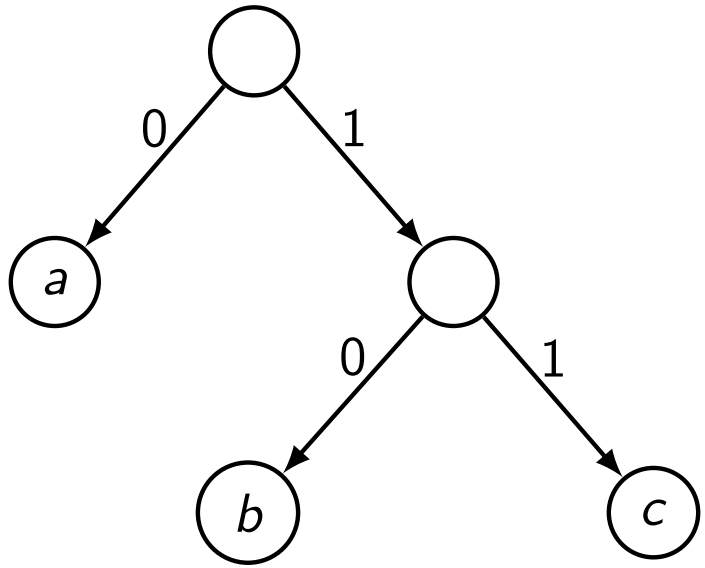
Dada una función f de frecuencias relativas para Σ y una Σ-codificación τ, el largo promedio de τ para f se define como:

$$\mathrm{lp}_f(\tau) \quad = \quad \sum_{a \in \Sigma} f(a) \cdot |\tau(a)|$$

La entrada del problema es entonces una función f de frecuencias relativas para Σ, y la función objetivo a minimizar es $\mathrm{lp}_f(x)$

Un ingrediente fundamental: codificaciones como árboles

Representamos la codificación $\tau(a) = 0$, $\tau(b) = 10$, $\tau(c) = 11$ como un árbol binario:



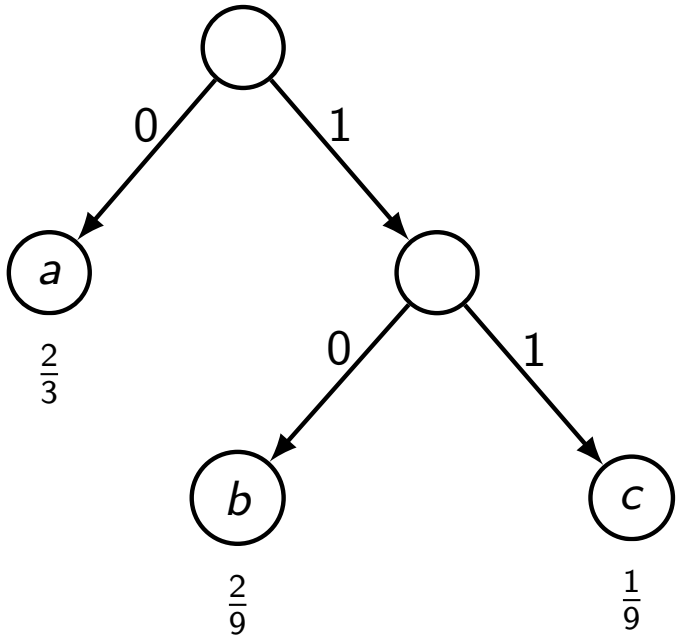
Un ingrediente fundamental: codificaciones como árboles

Si una Σ-codificación τ es libre de prefijos, entonces el árbol que la representa satisface las siguientes propiedades.

- Cada hoja tiene como etiqueta un elemento de Σ, y estos son los únicos nodos con etiquetas
- Cada símbolo de Σ es usado exactamente una vez como etiqueta
- Cada arco tiene etiqueta 0 ó 1
- Si una hoja tiene etiqueta e y las etiquetas de los arcos del camino desde la raíz hasta esta hoja forman una palabra $w \in \{0,1\}^*$, entonces $\tau(e) = w$

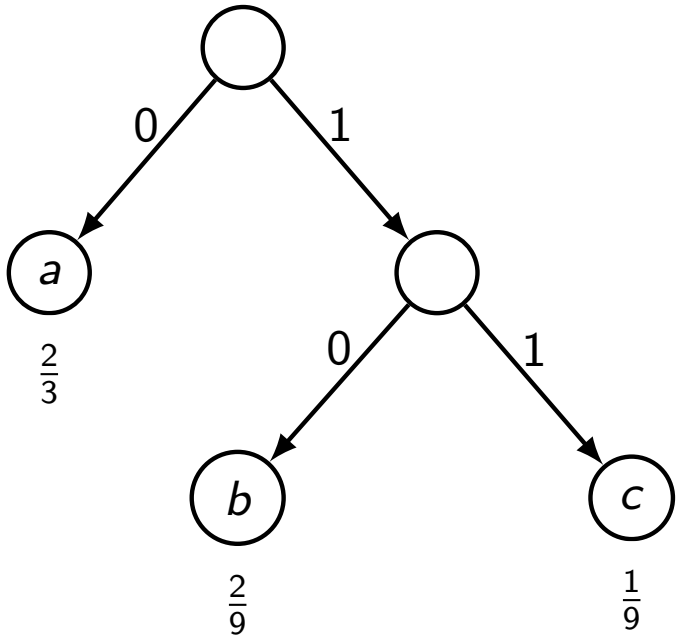
Codificaciones como árboles y las frecuencias relativas

Podemos agregar al árbol binario que representa una codificación τ la información sobre las frecuencias relativas dadas por una función f:



Codificaciones como árboles y las frecuencias relativas

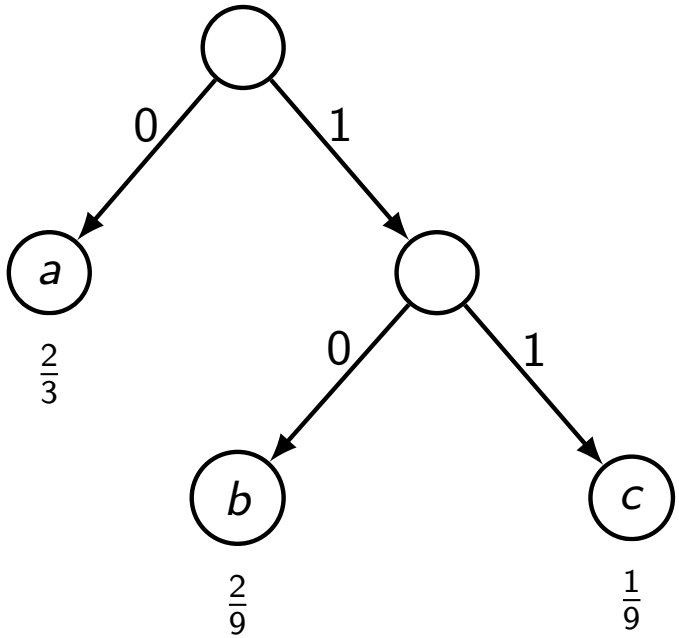
Podemos agregar al árbol binario que representa una codificación τ la información sobre las frecuencias relativas dadas por una función f:



Llamamos a este árbol abf(τ, f)

Codificaciones como árboles y las frecuencias relativas

Podemos agregar al árbol binario que representa una codificación τ la información sobre las frecuencias relativas dadas por una función f:



Llamamos a este árbol abf(τ, f)

- ¡Estos árboles son muy útiles!

Alguna propiedades importantes

Ejercicios

Sea f una función de frecuencias relativas para Σ y τ una Σ-codificación libre de prefijos que minimiza la función $\mathrm{lp}_f(x)$

1. Sean u y v dos hojas en $\mathrm{abf}(\tau, f)$ con etiquetas a y b, respectivamente. Demuestre que si el camino de la raíz a u es más corto que el camino de la raíz a v, entonces $f(a) \geq f(b)$

2. Demuestre que cada nodo interno en $\mathrm{abf}(\tau, f)$ tiene dos hijos

3. Sean $a, b \in \Sigma$ tales que $a \neq b$, $f(a) \leq f(b)$ y $f(b) \leq f(e)$ para todo $e \in (\Sigma \setminus \{a, b\})$. Demuestre que existe una Σ-codificación τ' libre de prefijos tal que $\mathrm{lp}_f(\tau') = \mathrm{lp}_f(\tau)$ y las hojas con etiquetas a y b en $\mathrm{abf}(\tau', f)$ son hermanas
   - Vale decir, existe $w \in \{0,1\}^*$ tal que $\tau'(a) = w0$ y $\tau'(b) = w1$

Calculando el mínimo de $\mathrm{lp}_f(x)$

Vamos a ver un algoritmo codicioso para calcular una Σ-codificación τ libre de prefijos que minimiza $\mathrm{lp}_f(x)$

- El algoritmo calcula la codificación de Huffman

Calculando el mínimo de $\mathrm{lp}_f(x)$

Vamos a ver un algoritmo codicioso para calcular una Σ-codificación τ libre de prefijos que minimiza $\mathrm{lp}_f(x)$

- El algoritmo calcula la codificación de Huffman

El algoritmo tiene los ingredientes mencionados de un algoritmo codicioso.

- Función objetivo a minimizar: $\mathrm{lp}_f(x)$
- Función de selección: elige los dos símbolos de Σ con menor frecuencia relativa, los coloca como hermanos en el árbol binario que representa la Σ-codificación óptima, y continua la construcción con el resto de los símbolos de Σ

Calculando el mínimo de $\mathrm{lp}_f(x)$

Vamos a ver un algoritmo codicioso para calcular una Σ-codificación τ libre de prefijos que minimiza $\mathrm{lp}_f(x)$

- El algoritmo calcula la codificación de Huffman

El algoritmo tiene los ingredientes mencionados de un algoritmo codicioso.

- Función objetivo a minimizar: $\mathrm{lp}_f(x)$
- Función de selección: elige los dos símbolos de Σ con menor frecuencia relativa, los coloca como hermanos en el árbol binario que representa la Σ-codificación óptima, y continua la construcción con el resto de los símbolos de Σ

Además, es necesario realizar una demostración para probar que el algoritmo es correcto.

El algoritmo de Huffman

En el siguiente algoritmo representamos a las funciones como conjuntos de pares ordenados, y suponemos que f es una función de frecuencias relativas que al menos tiene dos elementos en el dominio.

El algoritmo de Huffman

En el siguiente algoritmo representamos a las funciones como conjuntos de pares ordenados, y suponemos que f es una función de frecuencias relativas que al menos tiene dos elementos en el dominio.

CalcularCodificaciónHuffman(f)

Sea Σ el dominio de la función f

if $\Sigma = \{a, b\}$ **then return** $\{(a, 0), (b, 1)\}$

else

Sean $a, b \in \Sigma$ tales que $a \neq b$, $f(a) \leq f(b)$ y
$f(b) \leq f(e)$ para todo $e \in (\Sigma \setminus \{a, b\})$

Sea c un símbolo que **no** aparece en Σ

$g := (f \setminus \{(a, f(a)), (b, f(b))\}) \cup \{(c, f(a) + f(b))\}$

$\tau^\star :=$ **CalcularCodificaciónHuffman**(g)

$w := \tau^\star(c)$

$\tau := (\tau^\star \setminus \{(c, w)\}) \cup \{(a, w0), (b, w1)\}$

return τ

La correctitud de **CalcularCodificaciónHuffman**

Teorema

*Si f es una función de frecuencias relativas, Σ es el dominio de f y $\tau =$ **CalcularCodificaciónHuffman**(f), entonces τ es una Σ-codificación libre de prefijos que minimiza la función $lp_f(x)$*

La correctitud de **CalcularCodificaciónHuffman**

Teorema

*Si f es una función de frecuencias relativas, Σ es el dominio de f y $\tau =$ **CalcularCodificaciónHuffman**(f), entonces τ es una Σ-codificación libre de prefijos que minimiza la función $lp_f(x)$*

Demostración: Vamos a realizar la demostración por inducción en $|\Sigma|$

La correctitud de **CalcularCodificaciónHuffman**

Teorema

*Si f es una función de frecuencias relativas, Σ es el dominio de f y $\tau =$ **CalcularCodificaciónHuffman**(f), entonces τ es una Σ-codificación libre de prefijos que minimiza la función $lp_f(x)$*

Demostración: Vamos a realizar la demostración por inducción en $|\Sigma|$

Si $|\Sigma| = 2$, entonces la propiedad se cumple trivialmente

- ¿Por qué?

La correctitud de **CalcularCodificaciónHuffman**

Teorema

*Si f es una función de frecuencias relativas, Σ es el dominio de f y $\tau =$ **CalcularCodificaciónHuffman**(f), entonces τ es una Σ-codificación libre de prefijos que minimiza la función $lp_f(x)$*

Demostración: Vamos a realizar la demostración por inducción en $|\Sigma|$

Si $|\Sigma| = 2$, entonces la propiedad se cumple trivialmente

- ¿Por qué?

Suponga entonces que la propiedad se cumple para un valor $n \geq 2$, y suponga que $|\Sigma| = n + 1$

La demostración del teorema

Sean a, b, c, g, $\tau^\star$ y τ definidos como en el código de la llamada **CalcularCodificaciónHuffman**(f), y sea Γ el dominio de g

Como $|\Gamma| = n$ y $\tau^\star =$ **CalcularCodificaciónHuffman**(g), por hipótesis de inducción tenemos que $\tau^\star$ es una Γ-codificación libre de prefijos que minimiza la función $\mathrm{lp}_g(x)$

La demostración del teorema

Sean a, b, c, g, $\tau^\star$ y τ definidos como en el código de la llamada **CalcularCodificaciónHuffman**(f), y sea Γ el dominio de g

Como $|\Gamma| = n$ y $\tau^\star =$ **CalcularCodificaciónHuffman**(g), por hipótesis de inducción tenemos que $\tau^\star$ es una Γ-codificación libre de prefijos que minimiza la función $\mathrm{lp}_g(x)$

Dada la definición de τ es simple verificar las siguientes propiedades:

- τ es una codificación libre de prefijos
- Para cada $e \in (\Sigma \setminus \{a, b\})$ se tiene que $\tau(e) = \tau^\star(e)$
- $|\tau(a)| = |\tau(b)| = |\tau^\star(c)| + 1$

La demostración del teorema

Por contradicción suponga que τ no minimiza el valor de la función $\mathrm{lp}_f(x)$

- Vale decir, existe una Σ-codificación τ' libre de prefijos tal que τ' minimiza la función $\mathrm{lp}_f(x)$ y $\mathrm{lp}_f(\tau') < \mathrm{lp}_f(\tau)$

La demostración del teorema

Por contradicción suponga que τ no minimiza el valor de la función $\mathrm{lp}_f(x)$

- Vale decir, existe una Σ-codificación τ' libre de prefijos tal que τ' minimiza la función $\mathrm{lp}_f(x)$ y $\mathrm{lp}_f(\tau') < \mathrm{lp}_f(\tau)$

Por la definición de a, b y los ejercicios anteriores podemos suponer que existe $w \in \{0,1\}^*$ tal que $\tau'(a) = w0$ y $\tau'(b) = w1$

- Nótese que $w \neq \varepsilon$ puesto que $|\Sigma| \geq 3$

La demostración del teorema

Por contradicción suponga que τ no minimiza el valor de la función $\mathrm{lp}_f(x)$

- Vale decir, existe una Σ-codificación τ' libre de prefijos tal que τ' minimiza la función $\mathrm{lp}_f(x)$ y $\mathrm{lp}_f(\tau') < \mathrm{lp}_f(\tau)$

Por la definición de a, b y los ejercicios anteriores podemos suponer que existe $w \in \{0,1\}^*$ tal que $\tau'(a) = w0$ y $\tau'(b) = w1$

- Nótese que $w \neq \varepsilon$ puesto que $|\Sigma| \geq 3$

A partir de τ' defina la siguiente Γ-codificación τ'':

$$\tau'' \quad = \quad (\tau' \setminus \{(a, \tau'(a)), (b, \tau'(b))\}) \cup \{(c, w)\}$$

La demostración del teorema

Por contradicción suponga que τ no minimiza el valor de la función $\mathrm{lp}_f(x)$

- Vale decir, existe una Σ-codificación τ' libre de prefijos tal que τ' minimiza la función $\mathrm{lp}_f(x)$ y $\mathrm{lp}_f(\tau') < \mathrm{lp}_f(\tau)$

Por la definición de a, b y los ejercicios anteriores podemos suponer que existe $w \in \{0,1\}^*$ tal que $\tau'(a) = w0$ y $\tau'(b) = w1$

- Nótese que $w \neq \varepsilon$ puesto que $|\Sigma| \geq 3$

A partir de τ' defina la siguiente Γ-codificación τ'':

$$\tau'' \quad = \quad (\tau' \setminus \{(a, \tau'(a)), (b, \tau'(b))\}) \cup \{(c, w)\}$$

Tenemos que τ'' es una Γ-codificación libre de prefijos

- ¿Por qué?

La relación entre $\mathrm{lp}_g(\tau^\star)$ y $\mathrm{lp}_f(\tau)$

$$
\begin{aligned}
\mathrm{lp}_g(\tau^\star) &= \sum_{e \in \Gamma} g(e) \cdot |\tau^\star(e)| \\
&= \left(\sum_{e \in (\Gamma \setminus \{c\})} g(e) \cdot |\tau^\star(e)| \right) + g(c) \cdot |\tau^\star(c)| \\
&= \left(\sum_{e \in (\Sigma \setminus \{a,b\})} f(e) \cdot |\tau(e)| \right) + (f(a) + f(b)) \cdot |\tau^\star(c)| \\
&= \left(\sum_{e \in (\Sigma \setminus \{a,b\})} f(e) \cdot |\tau(e)| \right) + f(a) \cdot |\tau^\star(c)| + f(b) \cdot |\tau^\star(c)| \\
&= \left(\sum_{e \in (\Sigma \setminus \{a,b\})} f(e) \cdot |\tau(e)| \right) + f(a) \cdot (|\tau(a)| - 1) + f(b) \cdot (|\tau(b)| - 1) \\
&= \left(\sum_{e \in \Sigma} f(e) \cdot |\tau(e)| \right) - (f(a) + f(b)) \\
&= \mathrm{lp}_f(\tau) - (f(a) + f(b))
\end{aligned}
$$

La relación entre $\mathrm{lp}_g(\tau'')$ y $\mathrm{lp}_f(\tau')$

$$
\begin{aligned}
\mathrm{lp}_g(\tau'') &= \sum_{e \in \Gamma} g(e) \cdot |\tau''(e)| \\
&= \left(\sum_{e \in (\Gamma \setminus \{c\})} g(e) \cdot |\tau''(e)| \right) + g(c) \cdot |\tau''(c)| \\
&= \left(\sum_{e \in (\Sigma \setminus \{a,b\})} f(e) \cdot |\tau'(e)| \right) + (f(a) + f(b)) \cdot |\tau''(c)| \\
&= \left(\sum_{e \in (\Sigma \setminus \{a,b\})} f(e) \cdot |\tau'(e)| \right) + f(a) \cdot |\tau''(c)| + f(b) \cdot |\tau''(c)| \\
&= \left(\sum_{e \in (\Sigma \setminus \{a,b\})} f(e) \cdot |\tau'(e)| \right) + f(a) \cdot (|\tau'(a)| - 1) + f(b) \cdot (|\tau'(b)| - 1) \\
&= \left(\sum_{e \in \Sigma} f(e) \cdot |\tau'(e)| \right) - (f(a) + f(b)) \\
&= \mathrm{lp}_f(\tau') - (f(a) + f(b))
\end{aligned}
$$

Obteniendo una contradicción

Tenemos entonces que

$$\begin{aligned} \mathrm{lp}_g(\tau'') &= \mathrm{lp}_f(\tau') - (f(a) + f(b)) \\ &< \mathrm{lp}_f(\tau) - (f(a) + f(b)) \\ &= \mathrm{lp}_g(\tau^\star) \end{aligned}$$

Obteniendo una contradicción

Tenemos entonces que

$$\begin{aligned} \mathrm{lp}_g(\tau'') &= \mathrm{lp}_f(\tau') - (f(a) + f(b)) \\ &< \mathrm{lp}_f(\tau) - (f(a) + f(b)) \\ &= \mathrm{lp}_g(\tau^\star) \end{aligned}$$

Concluimos entonces que $\tau^\star$ no minimiza la función $\mathrm{lp}_g(x)$, lo cual contradice la hipótesis de inducción. □

Dos comentarios finales

La siguiente función calcula la codificación de Huffman teniendo como entrada una palabra w:

CalcularCodificaciónHuffman(w)
    if $w = \varepsilon$ **then return** $\emptyset$
    else
        $\Sigma :=$ conjunto de símbolos mencionados en w
        if $\Sigma = \{a\}$ **then return** $\{(a, 0)\}$
        else return CalcularCodificaciónHuffman(fr_w)

Dos comentarios finales

La siguiente función calcula la codificación de Huffman teniendo como entrada una palabra w:

CalcularCodificaciónHuffman(w)
    if $w = \varepsilon$ **then return** $\emptyset$
    else
        $\Sigma :=$ conjunto de símbolos mencionados en w
        if $\Sigma = \{a\}$ **then return** $\{(a, 0)\}$
        else return CalcularCodificaciónHuffman(fr_w)

¿Cuál es la complejidad de **CalcularCodificaciónHuffman**(f)?

- ¿Qué estructura de datos debería ser utilizada para almacenar f?