

Algoritmos y Complejidad

Programación Dinámica

Pablo R. Fillottrani

Depto. Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

Primer Cuatrimestre 2014



Programación Dinámica

Introducción

Problema de la mochila

Caminos más Cortos

Multiplicación de matrices en cadena

Triangulación optimal de polígonos

Subsecuencia común más larga



Introducción

- ▶ **Programación Dinámica** (PD) resuelve problemas a través de combinar soluciones a subproblemas
- ▶ PD comienza resolviendo las instancias más simples de los problemas, y guardando sus resultados en alguna estructura de datos especial
- ▶ para construir soluciones de instancias más complejas, se divide la instancia en subproblemas más simples y se recuperan los resultados ya calculados de la estructura de datos
- ▶ PD se aplica cuando los subproblemas **no son independientes** entre sí, es decir los subproblemas tienen subsubproblemas en común. Esto se denomina **superposición de subproblemas**

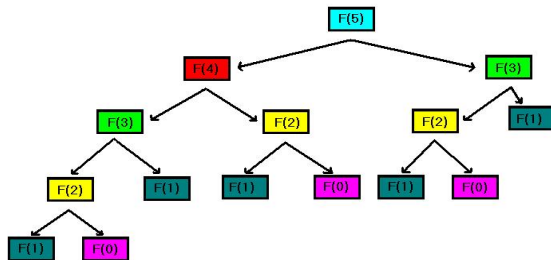


Comparación

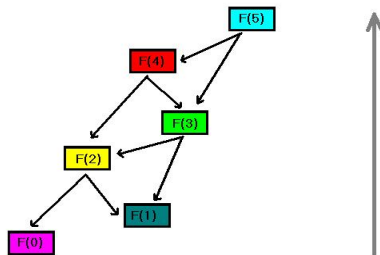
- ▶ veremos más adelante otra estrategia, “dividir y conquistar” (DYC), que resolvería las instancias siempre dividiendo, sin importar cálculos previos. En este contexto, resolvería **varias veces** el mismo subproblema. El algoritmo recursivo para calcular el número de Fibonacci tiene este inconveniente
- ▶ DYC se usa cuando **no hay superposición de subproblemas**, o es casi nula
- ▶ un algoritmo de PD resuelve cada subproblema una vez y guarda su resultado en una tabla, **evitando el trabajo de calcularlo otra vez**
- ▶ entonces para que se aplique PD tiene que ser eficiente (en tiempo y espacio) almacenar resultados de subproblemas previamente resueltos
- ▶ DYC es una técnica **top-down**; PD por el contrario es **bottom-up**



- Árbol de subproblemas del algoritmo DYC de Fibonacci para $n = 5$



- Árbol de subproblemas del algoritmo PD de Fibonacci para $n = 5$



- ▶ PD se aplica generalmente a problemas de **optimización**, al igual que los algoritmos *greedy*.
- ▶ pasos en el desarrollo de un algoritmo PD:
 1. caracterizar la **estructura** de una solución optimal
 2. definir **recursivamente** el valor de la solución optimal
 3. computar el valor de las soluciones a los **casos básicos**
 4. construir las **soluciones optimales** para instancias grandes a partir de la soluciones ya computadas para instancias más pequeñas

Elementos necesarios para aplicar PD

- ▶ **principio de optimalidad** la estructura de una solución optimal a un problema debe contener soluciones optimales a los subproblemas
- ▶ aunque parezca obvio, no todos los problemas satisfacen este principio (por ejemplo, el camino simple más largo entre dos nodos de un grafo)
- ▶ **superposición de subproblemas** el “espacio” de subproblemas debe ser pequeño en el sentido de que los subproblemas se repiten una y otra vez, en vez de generar nuevos subproblemas
- ▶ PD generalmente toma ventaja de esta repetición solucionando **una única vez cada subproblema**



Coeficientes Binomiales



$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0 \text{ o } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \\ 0 & \text{sino} \end{cases}$$

- como el caso base suma de a 1, el algoritmo recursivo directo tiene $\Omega\left(\binom{n}{k}\right)$

- ▶ no se trata de un problema de optimización, pero la solución está formada por combinación de soluciones de subproblemas
- ▶ además, claramente se ve superposición de subinstancias:

$$\begin{aligned}C(5,3) &= C(4,3) + C(4,2) = \\&= (C(3,3) + C(3,2)) + (C(3,2) + C(3,1)) = \dots\end{aligned}$$

- ▶ se puede suponer que es posible aplicar PD al problema.
- ▶ se puede usar una **tabla** para guardar resultados intermedios, donde la entrada (i,j) guarda el número $C(i,j)$

► se tiene

	0	1	2	3	4	...	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
...								
$n-1$							$C(n-1, k-1)$	$C(n-1, k)$
n								$C(n, k)$

► esta tabla se llama **triángulo de Pascal**, o **triángulo de Tartaglia**.



- el algoritmo para calcularla por **filas** es:

```
function CoeficientesBinomiales(n,k)
  array C[1..n,1..n]
  para todo k  C[k,0] ::= 1; C[k,k] ::= 1;
  FOR i ::= 0 TO n
    FOR j ::= 0 TO min(i,k)
      C[i,j] ::= C[i-1,j-1]+C[i-1,j]
    ENDFOR
  ENDFOR
  RETURN C[n,k]
```



Análisis del tiempo de ejecución

- ▶ su tiempo y espacio es claramente de $\Theta(nk)$.
- ▶ se puede modificar el algoritmo para que sólo use espacio $\Theta(k)$
(ejercicio)

Probabilidad de ganar una serie

- ▶ Problema: dos equipos A y B deben jugar hasta $2n - 1$ juegos, siendo el ganador el primer equipo que llega a n victorias. Para cada juego existe una probabilidad p de que gane el equipo A , y una probabilidad $q = 1 - p$ de que gane el equipo B . Esta probabilidad es fija para todos los juegos, e independiente de los resultados anteriores. Se quiere encontrar la probabilidad de que el equipo A gane la serie
- ▶ se define $P(i, j)$ como la probabilidad de que A gane la serie dado que le faltan i victorias, mientras que a B le faltan j victorias
- ▶ entonces el valor buscado es $P(n, n)$



- la formulación de esta propiedad genera la recurrencia:

$$P(i,j) = \begin{cases} 1 & \text{si } i = 0 \text{ y } j > 0 \\ 0 & \text{si } j = 0 \text{ y } i > 0 \\ pP(i-1,j) + qP(i,j-1) & \text{si } i > 0 \text{ y } j > 0 \end{cases}$$

- ▶ sea $k = j + i$. El algoritmo de cálculo recursivo de P tomaría tiempo:

$$T(1) = c$$

$$T(k) \leq 2T(k-1) + d$$

- ▶ la solución (usando la ecuación característica) es de $O(2^k)$, lo que equivale a $O(4^n)$ si $i = j = n$
- ▶ esta estructura del problema es similar a la de los coeficientes binomiales tomando $P(i, j)$ como $C(i+j, j)$

- ▶ es posible mejorar este tiempo en forma similar al triángulo de Pascal, calculando P por **filas**, **columnas** o **diagonales**
- ▶ para la cota inferior, da un tiempo de $\Omega\left(\binom{2n}{n}\right) \geq \frac{4^n}{n}$

- la matriz P resultaría:

	0	1	2	...	$j-1$	j
0	—	1	1		1	1
1	0	p	$p + pq$			
2	0	p^2	$p^2 + 2p^2q$			
3	0	p^3	$p^3 + 3p^3q$			
...						
$i-1$	0					$P(i-1, j)$
i	0				$P(i, j-1)$	$P(i, j)$

- esto demuestra la aplicación del principio de optimalidad en el problema
- se pueden calcular los elementos de la matriz por diagonales



```
function Serie(n,p)
  array P[0..n,0..n]
  FOR s ::= 1 TO n
    P[0,s] ::= 1; P[s,0] ::= 0
    FOR k ::= 1 TO s-1
      P[k,s-k] ::= p*P[k-1,s-k]+(1-p)*P[k,s-k-1]
    ENDFOR
  ENDFOR
  FOR s ::= 1 TO n
    FOR k ::= 0 TO n-s
      P[s+k,n-k] ::= p*P[s+k-1,n-k]+
                    (1-p)*P[s+k,n-k-1]
    ENDFOR
  ENDFOR; RETURN P[n,n]
```



Análisis del tiempo de ejecución

- ▶ su tiempo y espacio es de $\Theta(n^2)$
- ▶ se puede hacer la misma modificación que en el caso anterior para que use espacio en $\Theta(n)$

Problema del Cambio

- ▶ Problema: se tiene que dar N centavos de cambio, usando la menor cantidad entre monedas de denominaciones $d_1, d_2, d_3, \dots, d_n$. Se supone cantidad ilimitada de monedas de cada denominación
- ▶ el algoritmo *greedy* visto sólo es correcto para ciertas denominaciones; en otras puede que ni siquiera encuentre una solución a pesar de que ésta exista
- ▶ para definir un algoritmo de PD para este problema, se define $C[i, j]$ la menor cantidad de monedas entre d_1, d_2, \dots, d_i para pagar j centavos



- ▶ la solución está entonces $C[n, N]$
- ▶ una de las dimensiones de la matriz es el conjunto de denominaciones usadas; esto es usual en problemas de PD donde existe una secuencia de objetos a considerar
- ▶ se satisface el **principio de optimalidad** Si la solución optimal $C[n, N]$ incluye una moneda de d_n entonces deberá estar formada por la solución optimal $C[n, N - d_n]$. En cambio si no incluye ninguna moneda de d_n , su valor será la solución optimal a $C[n - 1, N]$

- la recurrencia quedaría:

$$C[i,j] = \begin{cases} 0 & \text{si } j = 0 \\ +\infty & \text{si } i = 1 \text{ y } 0 < j < d_i \\ 1 + C[i, j - d_i] & \text{si } i = 1 \text{ y } j \geq d_i \\ C[i - 1, j] & \text{si } i > 1 \text{ y } j < d_i \\ \min(C[i - 1, j], 1 + C[i, j - d_i]) & \text{si } i > 1 \text{ y } j \geq d_i \end{cases}$$

- por ejemplo para $N = 8$ con $d_1 = 1$, $d_2 = 4$ y $d_3 = 6$ se tiene:

Centavos	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

Algoritmo

```
function Cambio(D[1..n],N)
  array C[1..n,0..N]
  FOR i:=1 TO n; C[i,0]:=0;
    FOR j:=1 TO N
      CASE
        i=1 y j<d[i] : C[i,j]:=+maxint
        i=1 y j>=d[i] : C[i,j]:=1+C[i,j-d[i]]
        i>1 y j<d[i] : C[i,j]:=C[i-1,j]
        i>1 y j>=d[i] : C[i,j]:=min(C[i-1,j],
                                   1+C[i,j-d[i]])
      ENDCASE
    ENDFOR
  ENDFOR; RETURN C[n,N]
```



- ▶ el tiempo y el espacio es de $\Theta(nN)$
- ▶ este algoritmo sólo encuentra el **mínimo número de monedas** necesarios, pero no dice cuáles son
- ▶ para encontrar las monedas que forman el cambio, **se analiza cada la entrada $C[i, j]$** : si es igual a $C[i - 1, j]$ entonces no se usan monedas d_i ; en caso contrario se usa una moneda d_i más las monedas de $C[i, j - d_i]$
- ▶ partiendo de $C[n, N]$ y retrocediendo por fila, o por columna, de acuerdo a su valor, hasta llegar a $C[0, 0]$, se obtienen las $C[n, N]$ monedas que forman el cambio
- ▶ este recorrido agrega tareas por tiempo $\Theta(n + C[n, N])$ al algoritmo original



- ▶ **Observación:** la dependencia del tiempo y el espacio de ejecución en un dato de entrada N no es buena porque puede ser arbitrariamente grande
- ▶ ¿cómo se modificaría el programa si se dispone de una cantidad limitada de monedas de cada denominación? (**ejercicio**)

Definición del problema

- Problema: se tienen n objetos indivisibles y una mochila. Cada objeto i tiene un peso w_i y un valor v_i ; la mochila tiene una capacidad máxima de W . El objetivo es encontrar la carga de la mochila que maximice el valor de lo transportado y se respete su capacidad máxima
- es decir, encontrar valores $x_i = 0, 1$, de forma que

$$\text{maximice } \sum_{i=1}^n x_i v_i \quad \text{siempre que } \sum_{i=1}^n x_i w_i \leq W$$

- en esta variante no se permite fraccionar los objetos (**ejercicio:** mostrar que el algoritmo *greedy* visto anteriormente no es correcto en este caso)



- ▶ para aplicar PD a este problema basta con mostrar que cumple con el **principio de optimalidad** y que tiene superposición de subinstancias
- ▶ la función a optimizar es el valor de la carga de la mochila. Este valor depende de W y de la cantidad de objetos considerados
- ▶ sea entonces $V[i, j]$ el máximo valor de una carga de peso a lo sumo j con los objetos $1, 2, \dots, i$
- ▶ al igual que en el caso del problema del cambio, una de las dimensiones es el conjunto de objetos

- ▶ el valor de $V[i, j]$ depende de si se incluye o no el objeto i .
- ▶ la recurrencia es

$$V[i, j] = \begin{cases} 0 & \text{si } i = 0 \\ -\infty & \text{si } i > 0 \text{ y } j < 0 \\ \max(V[i-1, j], v_i + V[i-1, j-w_i]) & \text{si } i > 0 \text{ y } j > 0 \end{cases}$$

- ▶ la dependencia es con elementos de filas anteriores, a lo sumo en la misma columna

Ejemplo

► por ejemplo, si $W = 11$

Peso, Valor	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1, v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2, v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5, v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6, v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7, v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

Algoritmo

- ▶ el algoritmo para implementar este algoritmo es muy similar al algoritmo para el problema del cambio
- ▶ el tiempo y el espacio es de $\Theta(nW)$
- ▶ para calcular cuáles objetos componen la carga optimal se puede hacer un recorrido adicional desde $C[n, W]$ hasta $C[0, 0]$ de $\Theta(n + W)$

Definición del problema

- ▶ Problema: Sea $G = \langle N, A \rangle$ un grafo dirigido, con pesos no negativos. El objetivo es hallar el camino con la mínima distancia entre cada par de nodos. Supondremos el grafo representado por una matriz de adyacencia
- ▶ una solución a este problema consiste en ejecutar ***n veces el algoritmo de Dijkstra*** cambiando en cada iteración el nodo origen
- ▶ vale el ***principio de optimalidad*** en este caso: si k es un nodo en el menor camino entre i y j , entonces ese camino está formado por el menor camino de i a k y el menor camino de k a j
- ▶ también hay superposición de instancias



- ▶ para aplicar PD a este problema se debe ir construyendo la menor distancia a partir de ir agregando nodos a los caminos posibles
- ▶ sea entonces $D[i,j,k]$ la menor distancia entre i y j que tiene como nodos intermedios a $1, 2, \dots, k$. Los valores buscados serán entonces $D[i,j,n]$
- ▶ en cada iteración, para cada par (i,j) se debe comparar el camino más corto obtenido hasta entonces con el camino que va desde i hasta k , y luego de k a j , pasando sólo por los nodos $1, \dots, k-1$
- ▶ se tiene en cuenta implícitamente el hecho de que **un camino optimal no puede pasar dos veces por un nodo**



- ▶ los valores iniciales, cuando $k = 0$, corresponde a los pesos de los arcos (i, j)
- ▶ la recurrencia es

$$D[i, j, k] = \begin{cases} G[i, j] & \text{si } k = 0 \\ \min(D[i, j, k-1], D[i, k, k-1] + D[k, j, k-1]) & \text{sino} \end{cases}$$

- ▶ resultando en un algoritmo en $\Theta(n^3)$



- ▶ el espacio del algoritmo anterior también es de $\Theta(n^3)$
- ▶ se puede **mejorar el espacio** para este cálculo teniendo en cuenta:
 - ▶ cada $D[i, j, k]$ sólo necesita conocer los valores en $D[*, *, k - 1]$.
Esto reduce el espacio a $\Theta(n^2)$
 - ▶ para todo k , $D[k, j, k] = D[k, j, k - 1]$ y $D[i, k, k] = D[i, k, k - 1]$.
Esto reduce el espacio a $\Theta(1)$ (sin contar el necesario para almacenar el resultado)
- ▶ el resultado es de $O(n^2)$



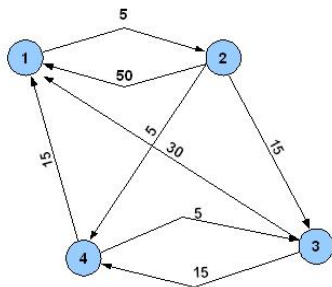
- el algoritmo resultante se llama **algoritmo de Floyd**, muy simple y conocido

```
function Floyd(G[1..n,1..n])  
  array D[1..n,1..n]  
  D::=G  
  FOR k::=1 TO n  
    FOR i::=1 TO n  
      FOR j::=1 TO n  
        D[i,j]:=min(D[i,j],D[i,k]+D[k,j])  
      ENDFOR  
    ENDFOR  
  ENDFOR  
  RETURN D
```



- ▶ su tiempo es de $\Theta(n^3)$ y el espacio es de $\Theta(1)$
- ▶ el tiempo es comparable con n veces Dijkstra, pero su **simplicidad** hace que se prefiera implementar Floyd

Ejemplo



$$D_0 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

- ▶ este algoritmo sólo encuentra las distancias mínimas entre cada par de nodos. Para obtener los nodos que implementan esa distancia es necesario recordar para cada (i, j) cuál es el k que proveyó la mínima distancia entre ellos
- ▶ es suficiente con actualizar una matriz adicional P cada vez que se modifica $D[i, j]$, reemplazando la línea interna de los FOR por

```
IF  $D[i, j] > D[i, k] + D[k, j]$   
   $D[i, j] ::= D[i, k] + D[k, j]$   
   $P[i, j] ::= k$   
ENDIF
```



- en el ejemplo visto se obtendría al finalizar

$$P = \begin{pmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Definición del problema

- ▶ Problema: se tienen n matrices M_1, M_2, \dots, M_n , no necesariamente cuadradas, y se quiere encontrar la mejor manera de hallar su producto $M_1 M_2 \dots M_n$. Cada matriz M_i es de tamaño $d_{i-1} d_i$
- ▶ teniendo en cuenta que:
 - ▶ cada producto $M_i M_{i+1}$ se calcula con $d_{i-1} d_i d_{i+1}$ productos escalares
 - ▶ el producto entre matrices es asociativo, luego
$$(M_i M_{i+1}) M_{i+2} = M_i (M_{i+1} M_{i+2})$$
- ▶ entonces es **relevante el orden** en que se realiza el producto M_1, M_2, \dots, M_n .
- ▶ ejemplo: M_1 de 5×10 , M_2 de 10×20 , M_3 de 20×2 , entonces $M_1 (M_2 M_3)$ lleva $400 + 100 = 500$ productos, y $(M_1 M_2) M_3$ lleva $1000 + 200 = 1200$ productos



- ▶ el problema entonces consiste en encontrar todas las **parentizaciones** posibles para M_1, M_2, \dots, M_n , evaluar la cantidad de productos necesarios, y obtener el menor entre todos ellos
- ▶ la cantidad de parentizaciones posibles está definida por la recurrencia

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i)$$

con $T(1) = 1$

- ▶ los $T(n)$ forman los llamados **números de Catalan** y se puede probar que $T(n) \in \Omega(4^n/n^2)$ por inducción
- ▶ luego el algoritmo directo toma tiempo de $\Omega(4^n/n)$ por lo que es inviable en la práctica para n medianos



- ▶ este problema satisface el principio de optimalidad
- ▶ y tiene también superposición de instancias
- ▶ es posible entonces aplicar PD



- ▶ la función a optimizar es la **cantidad de productos de reales** necesarios para multiplicar una secuencia de matrices. este valor depende de la cantidad de productos necesarios para multiplicar subsecuencias de matrices
- ▶ se define $m_{ij}, i \leq j$ como la mínima cantidad de productos necesarios para calcular $M_i \dots M_j$. Claramente, si $i = j$ entonces $m_{ii} = 0$ y si $j = i + 1$ entonces $m_{i,i+1} = d_{i-1} d_i d_{i+1}$
- ▶ en general, si $i < j$

$$m_{ij} = \min_{i \leq k < j} (m_{ik} + m_{(k+1)j} + d_{i-1} d_k d_j)$$



- por ejemplo, si $d = (10, 5, 20, 30, 2)$:

$i \backslash j$	1	2	3	4
1	0	1000	4500	1500
2		0	3000	1400
3			0	1200
4				0

$$\begin{aligned} m_{13} &= \min(m_{12} + m_{33} + d_0 d_2 d_3, m_{11} + m_{23} + d_0 d_1 d_3) = \\ &= \min(1000 + 6000, 3000 + 1500) = 4500 \end{aligned}$$

$$\begin{aligned} m_{24} &= \min(m_{23} + m_{44} + d_1 d_3 d_4, m_{22} + m_{34} + d_1 d_2 d_4) = \\ &= \min(3000 + 300, 1200 + 200) = 1400 \end{aligned}$$

$$\begin{aligned} m_{14} &= \min(m_{11} + m_{24} + d_0 d_1 d_4, m_{12} + m_{34} + d_0 d_2 d_4, \\ &\quad m_{13} + m_{44} + d_0 d_3 d_4) = \\ &= \min(1400 + 100, 1200 + 1000 + 400, 4500 + 600) = 1500 \end{aligned}$$



Algoritmo

```
function MultMatrices(d[0..n])  
  array m[1..n,1..n]::=0;  
  FOR s:=1 TO n-1  
    FOR i:=1 TO n-s; menor:= +maxint  
      FOR k:=i TO i+s-1  
        tmp:=m[i,k]+m[k+1,i+s]+d[i-1]*d[k]*d[i+s]  
        IF tmp<menor THEN menor:=tmp  
      ENDFOR  
      m[i,i+s]::=menor  
    ENDFOR  
  ENDFOR  
  RETURN m[1,n]
```



- el tiempo ejecución, tomando como barómetro cualquiera de la sentencias del ciclo interno, es:

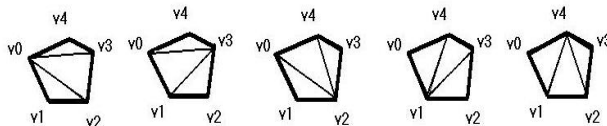
$$\begin{aligned}
 T(n) &= \sum_{s=1}^{n-1} \sum_{i=1}^{n-s} \sum_{k=i}^{i+s-1} c = \sum_{s=1}^{n-1} \sum_{i=1}^{n-s} sc = \\
 &= c \sum_{s=1}^{n-1} \sum_{i=1}^{n-s} s = c \sum_{s=1}^{n-1} (n-s)s = nc \sum_{s=1}^{n-1} s - c \sum_{s=1}^{n-1} s^2 = \\
 &= n \frac{c}{2} (n-1)n - (n-1)n(2n-1) \frac{c}{6} = \frac{c}{6} n^3 - \frac{c}{6} n \\
 &\in \Theta(n^3)
 \end{aligned}$$

- para obtener cuál es la mejor forma de multiplicar la matrices, es suficiente con recordar para cada (i, j) cuál es el k que determinó su menor valor (**ejercicio**).
- existen algoritmos más eficientes para este problema



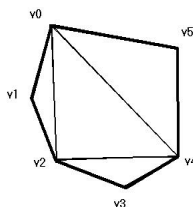
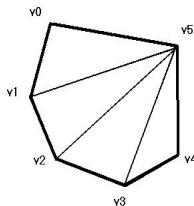
Definición del problema

- ▶ el algoritmo anterior tiene muchas aplicaciones, no directamente relacionadas con la multiplicación de matrices. Por ejemplo, para la **triangularización de polígonos**
- ▶ Problema: se tiene un polígono convexo $\langle v_0, v_1, \dots, v_{n-1} \rangle$ de n lados, siendo v_i los vértices y $\overline{v_{i-1}v_i}$ el lado i . Se quiere encontrar una triangulación optimal, de acuerdo a algún criterio dado



- ▶ una **cuerda** $\overline{v_i v_j}$ es el segmento formado por un par de vértice no adyacentes
- ▶ toda cuerda divide al polígono en dos subpolígonos
- ▶ una **triangularización** es un conjunto de cuerdas que dividen al polígono en triángulos disjuntos
- ▶ si se tiene un peso $w(\triangle v_i v_j v_k)$ para cada triángulo $\triangle v_i v_j v_k$, entonces una triangularización optimal de un polígono es una triangularización que minimiza la sumatoria de los pesos de los triángulos resultantes

- ▶ una función común para pesar los triángulos es su perímetro:
 $w(\triangle v_i v_j v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|$. Otras pueden usarse
- ▶ cada triangularización de un polígono de n lados consta de $n - 3$ cuerdas y $n - 2$ triángulos ([ejercicio](#))

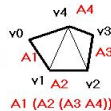
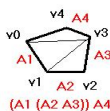
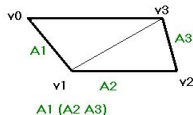
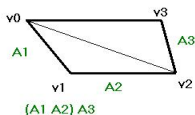


Reducción

- ▶ la estructura de este problema es similar a la de la **multiplicación cadena de matrices**
- ▶ se define una **reducción TRIANGULARIZACIÓN** \longrightarrow **CADENAMATRICES**
- ▶ dado un polígono $\langle v_0, v_1, \dots, v_{n-1} \rangle$, se establece una **correspondencia** entre los lados (excepto $\overline{v_{n-1} v_0}$) y “matrices” A_i , cuyo “tamaño” es $v_{i-1} v_i$ y con “tiempo de multiplicación” $w(\triangle v_i v_j v_k)$

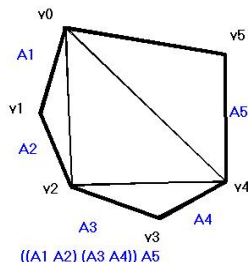
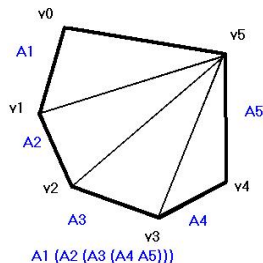


- luego cada forma de multiplicar las matrices $A_1 A_2 \dots A_{n-1}$ corresponde a una triangulación del polígono



- ▶ en el algoritmo, simplemente se reemplaza el costo de cada producto individual. Para las matrices era $d_i * d_j * d_k$, mientras que para la triangulación es $w(\Delta v_i v_j v_k)$

$temp ::= m[i, k] + m[k+1, i+s] + w(v[i], v[j], v[k])$



- ▶ en bioinformática, es frecuente la necesidad de comparar el ADN de dos o más organismos
- ▶ una secuencia de ADN se representa como una cadena en la letras que representan cada una de las bases posible: A (adenina), G (guanina), C (citosina) y T (tiamina). Ejemplo: ACCGGTCGGGATGCACCTGAGAAAGCGG
- ▶ un posible criterio de “similitud” entre secuencias de DNA es encontrar la **subsecuencia común más larga** de bases que aparezca en las secuencias aún en forma no consecutiva
- ▶ por ejemplo, para AGCGTAG y GTCAGA la subsecuencia común más larga es GCGA
- ▶ no es lo mismo que la subcadena más larga, ya que se permiten otros caracteres en el medio



Formalización del problema

- ▶ formalmente, dadas una secuencia $X = \langle x_1, x_2, \dots, x_m \rangle$, otra secuencia $Z = \langle z_1, z_2, \dots, z_k \rangle$ es una **subsecuencia** si existe una secuencia creciente de índices i_1, i_2, \dots, i_k tal que $x_{i_j} = z_j$ para todo $j, 1 \leq j \leq k$
- ▶ ejemplo, para $X = \langle A, G, C, G, T, A, G \rangle$, $Z = \langle G, C, T, G \rangle$ es una subsecuencia con índices 2, 3, 5, 7
- ▶ dadas dos secuencias X, Y se dice que Z es una **subsecuencia común** de X, Y si Z es subsecuencia de X y Z es subsecuencia de Y
- ▶ dadas dos secuencias X, Y el problema de la **subsecuencia común más larga (LCS)** es el problema de encontrar una subsecuencia común de longitud máxima para X, Y



Estructura optimal

- ▶ un algoritmo de fuerza bruta para resolver LCS es enumerar todas las posibles subsecuencias de X , controlar si también es subsecuencia de Y , y recordar la más larga de ellas
- ▶ este algoritmo es de $O(2^m)$, y por lo tanto inviable para m grandes
- ▶ sin embargo, es posible comprobar que LCS tiene una subestructura optimal

Subestructura optimal de LCS

Teorema

Sean $X = \langle x_1, x_2, \dots, x_m \rangle$ e $Y = \langle y_1, y_2, \dots, y_n \rangle$. Luego si $Z = \langle z_1, z_2, \dots, z_k \rangle$ es LCS de X, Y y

- ▶ $x_m = y_n$, entonces Z_{k-1} es LCS de X_{m-1}, Y_{n-1}
- ▶ $x_m \neq y_n$ y $z_k \neq x_m$, entonces Z es LCS de X_{m-1}, Y
- ▶ $x_m \neq y_n$ y $z_k \neq y_n$, entonces Z es LCS de X, Y_{n-1}

Demostración.

Se prueban los tres punto por contradicción, llegando en todos los casos a mostrar que Z no es LCS de X, Y .



Solución recursiva

- ▶ el teorema anterior sugiere la siguiente recurrencia para resolver LCS, siendo $C[i, j]$ el LCS de X_i, Y_j

$$C[i, j] = \begin{cases} 0 & \text{si } i = 0 \text{ o } j = 0 \\ C[i-1, j-1] + 1 & \text{si } i > 0, j > 0 \text{ y } x_i = y_j \\ \max(C[i-1, j], C[i, j-1]) & \text{si } i > 0, j > 0 \text{ y } x_i \neq y_j \end{cases}$$

- ▶ se puede ver fácilmente que existe superposición de problemas

Algoritmo PD

```
function LCS(X[1..m], Y[1..n])  
  array C[1..m,1..n]::=0  
  FOR i::=1 TO m  
    FOR j::=1 TO n  
      IF X[i]==Y[j]  
        C[i,j]::=C[i-1,j-1]+1  
      ELSIF C[i-1,j]>=C[i,j-1]  
        C[i,j]::=C[i-1,j]  
      ELSE  
        C[i,j]::=C[i,j-1]  
      ENDIF  
    ENDFOR  
  ENDFOR ; RETURN C[m,n]
```



Análisis del algoritmo

- ▶ el algoritmo anterior es de tiempo y espacio en $O(mn)$
- ▶ **Ejercicio:** retornar no sólo la longitud de la LCS entre dos cadenas, sino también una cadena que sea la LCS
- ▶ **Ejercicio:** ¿cómo modificaría el algoritmo para que compute **todas** las LCS entre dos cadenas, sin aumentar el orden del tiempo ni espacio?