Programación Dinámica

Esquema

1. Motivación y características
2. Elementos de Programación Dinámica
   - Principio de Optimalidad de Bellman
   - Definición Recursiva de la solución optimal
   - Enfoque ascendente
   - Búsqueda solución óptima
3. Resolución de problemas tipo
   - Devolver cambio
   - Mochila 0/1
   - Multiplicación de Matrices
   - Subsecuencia de longitud mayor (LCS)
   - Caminos mínimos

Motivación: Fibonacci

- La sucesión de Fibonacci es:
- F_n = n si n=0 o n=1
- $F_n = F_{n-1}+F_{n-2}$ si n>1
- Posible algoritmo de cálculo (recursivo)

```
función FIBONACCI ( n )
  si ( n <= 1 )
      devolver n
  si ( n > 1 )
      devolver FIBONACCI (n-1)+FIBONACCI (n-2)
```

Fibonacci

- Problema: no es eficiente porque se repiten muchos cálculos
- f6 = f5+f4 = f4+f3+f3+f2 = f3+f2+f2+f1+f2+f1+f2 = f2+f1+f2+f2+f1+f2+f1+f2
- f2 se calcula 5 veces!
- Los subproblemas se solapan
- Podríamos mejorar si almacenamos los resultados y calculamos cada fi una sola vez

```
fibDPwrap(n)
    Dict  soln = create(n);
    return fibDP(soln, n);

fibDP(soln, k)
    int  fib, f1 , f2;
    if (k < 2)
        fib = k;
    else
        if (member(soln, k-1) == false)
            f1 = fibDP(soln, k-1);
        else
            f1 = retrieve(soln, k-1);

        if (member(soln, k-2) == false)
            f2 = fibDP(soln, k-2);
        else
            f2 = retrieve(soln, k-2);

        fib = f1 + f2;
    store(soln, k, fib);
    return fib;
```

Fibonacci lineal

- Podemos construir un algoritmo lineal usando una tabla para almacenar los cálculos y organizando los cálculos de forma ordenada:

```
If n<=1 return n;
Else {
        T[0]=0; T[1]=1;
        For i=2 to n
           T[i]=T[i-1]+T[i-2];
       }
       return T[n];
```

Fibonacci aun mejor

- Podemos hacer el algoritmo más eficiente en espacio, ahorrándonos la tabla:

```
If n<=1 return n;
Else {
        x=1; y=0;
        For i=2 to n
           suma=x+y;
           y=x;
           x=suma;
       }
       return suma;
```

Números combinatorios

$$\binom{n}{k} = \frac{n!}{(n-k)!\;k!}$$

calcula el número de subconjuntos de tamaño k que se pueden formar con n elementos distintos.

Se pueden calcular recursivamente

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

```
funcion C(n,k)
  si k=0 o k=n devolver 1
  sino devolver C(n-1,k-1)+C(n-1,k)
```

Números combinatorios

- Problema: Muchos de los valores C(i,j), con $i<n, j<k$ se calculan varias veces
- C(5,3)=C(4,2)+C(4,3)=(C(3,1)+C(3,2))+(C(3,2)+C(3,3))=

  =C(2,0)+C(2,1)+C(2,1)+C(2,2)+C(2,1)+C(2,2)+1=
- =1+C(1,0)+C(1,1)+C(1,0)+C(1,1)+1+C(1,0)+C(1,1)+1+1=10

Números combinatorios

- Si utilizáramos una tabla de resultados intermedios C[i,j], i=1..n, j=0..k (el triángulo de Pascal) obtenemos un resultado más eficiente.
- Rellenando la tabla línea por línea:

	0	1	2	3	4	5	⋯
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
⋮			⋮				⋱

```
for i=1 to n
 for j=0 to k
  if j<=i
   if (j=0 or j=i) C[i,j]=1
   else C[i,j]=C[i-1,j-1]+C[i-1,j]
return C[n,k]
```

Números combinatorios

- Incluso podemos usar simplemente dos vectores de tamaño k (que representan la línea actual y la anterior) (o incluso un solo vector, actualizándolo de dcha. a izqda.)

- Este algoritmo requiere un tiempo O(nk) y un espacio O(k)

Motivación: Qué hemos visto

- Cuando un problema se divide recursivamente en subproblemas que se solapan, la eficiencia disminuye mucho.
- Se pueden mantener en memoria los subcasos resueltos para no repetir cálculos y mejorar la eficiencia.
- “Aquellos que no recuerdan su pasado están condenados a repetirlo”

Programación Dinámica: Características de los problemas

- Suelen ser problemas de optimización
- el problema debe poder resolverse por etapas, Sol=d1,d2,d3,...,dn, se toma una decisión en cada paso, pero esta depende de las soluciones a los subproblemas que lo componen
- debe poder modelizarse con una función recurrente
- debe cumplir el Principio de Optimalidad de Bellman

Programación Dinámica: Características de los problemas

- Esta técnica se aplica sobre problemas que a simple vista necesitan un alto coste computacional (posiblemente exponencial) donde:
  - **Subproblemas optimales**: La solución óptima a un problema puede ser definida en función de soluciones óptimas a subproblemas de tamaño menor, generalmente de forma recursiva.
  - **Solapamiento entre subproblemas:** Al plantear la solución recursiva, un mismo problema se resuelve más de una vez.

PD: Características de la técnica

- Introducida por Richard Bellman en 1957

- No tiene nada que ver con la programación (es más bien “planificación”)

- Resuelve un problema por etapas (como greedy o backtracking)

- Divide el problema en subproblemas (como Divide y Vencerás)

PD: Características de la técnica

- Suele ser una técnica ascendente (bottom-up) para obtener la solución, primero calcula las soluciones óptimas a problemas de tamaño pequeño. Utilizando dichas soluciones encuentra soluciones a problemas de mayor tamaño.

- Retiene en memoria las soluciones de los subproblemas, para evitar cálculos repetidos (memoizing)

- Devuelve la solución óptima (principio de Optimalidad de Bellman)

PD: Idea general

- **Encontrar una formulación recursiva de la solución de un problema mayor en función de soluciones a problemas menores.**
- **Resolver cada instancia de tamaño menor una única vez y guardarla en una tabla.**
- **Obtener la solución de la instancia inicial utilizando las soluciones almacenadas.**

PD: Análisis de la eficiencia

- Depende del problema, aunque suele ser polinomial

- En general, al utilizar una tabla, será $n \cdot m$
  - n es el tamaño de la tabla
  - m tiempo para rellenar cada casilla

- Algunos cálculos pueden ser innecesarios
- Puede dar problemas al necesitar mucha memoria (para rellenar la tabla)

PD versus Divide y Vencerás

Ambos combinan soluciones de subproblemas, pero

- DyV se aplica cuando los subproblemas son independientes
- PD se aplica cuando los subproblemas se solapan
- DyV repetiría muchos cálculos. PD los mantiene en memoria
- DyV utiliza un método descendente
- PD utiliza un método ascendente (normalmente)
- DyV utiliza recurrencias (+ tiempo, - memoria)
- PD intenta evitar recurrencias, utiliza memoria para iteración (- tiempo, + memoria)
- En ambos casos se obtiene la solución óptima

PD versus Greedy

Ambos resuelven el problema por etapas, pero

- Greedy selecciona un elemento y sólo genera una solución, en cada etapa
- PD selecciona un elemento en cada paso, pero genera múltiples caminos de etapas a seguir. Elige la optimal entre ellas
- Greedy no asegura optimalidad (miope)
- PD asegura optimalidad (Principio de Optimalidad de
- Bellman)
- Greedy es eficiente en tiempo y en memoria
- PD es eficiente en tiempo pero no en memoria

Principio de Optimalidad de Bellman

Una secuencia óptima de decisiones que resuelve un problema debe cumplir la propiedad de que cualquier subsecuencia de decisiones debe ser también óptima respecto al subproblema que resuelve.

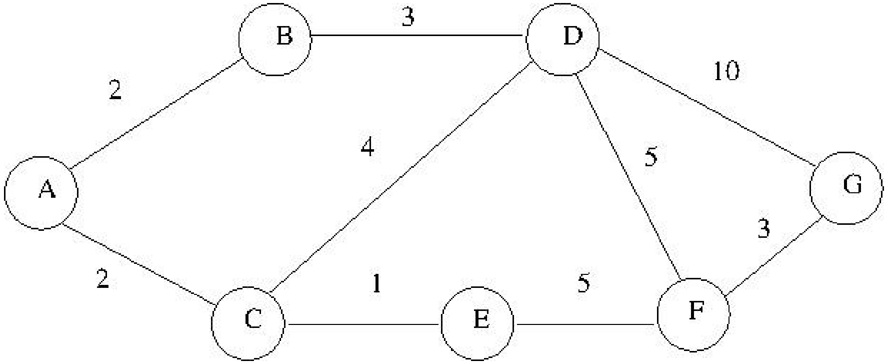
Esto es,

La solución óptima a cualquier caso no trivial de un problema es una combinación de soluciones óptimas de algunos de los subcasos.

Si d1,d2,d3, …, dn es optimal para P[1,n]

- *Entonces d1,d2,…,di es optimal para P[1,i]*
- *y di+1,…,dn es optimal para P[i+1,n]*

Principio de Optimalidad: caminos mínimos



Camino mínimo de A a G es ACEFG con valor 11
Otros caminos de A a G:

ACDFG 14
ACDG 16
ABDG 15
ABDFG 13

ACE (3) es el camino mínimo de A a E
EFG (8) es el camino mínimo de E a G
Subcaminos del camino mínimo también son mínimos para el subproblema correspondiente

Camino mínimo de A a D es ABD 5
Camino mínimo de D a G es DFG 8
Pero la composición de caminos mínimos ABD y DFG NO es el camino mínimo de A a G

Principio de optimalidad

- La dificultad en aplicar este principio está en que no suele ser evidente cuáles son los subcasos relevantes para el caso considerado.
- Esto impide usar una aproximación similar a divide y Vencerás, comenzando en el caso original y buscando recursivamente soluciones óptimas para los subcasos relevantes y sólo para estos.
- En su lugar la PD resuelve todos los subcasos, para determinar los que realmente son relevantes; y entonces se combinan en una solución óptima para el caso original.

Pasos para desarrollar un algoritmo basado en PD

1. Planteamiento de la solución como una sucesión de decisiones y verificación de que se cumple el Principio de Optimalidad de Bellman:
   - Encontrar la estructura de la solución:
     - Dividir el problema en subproblemas y determinar si se puede aplicar el principio de optimalidad.
2. Definición recursiva de la solución optimal:
   - Definir el valor de la solución óptima en función de valores de soluciones para sub-problemas de tamaño menor.

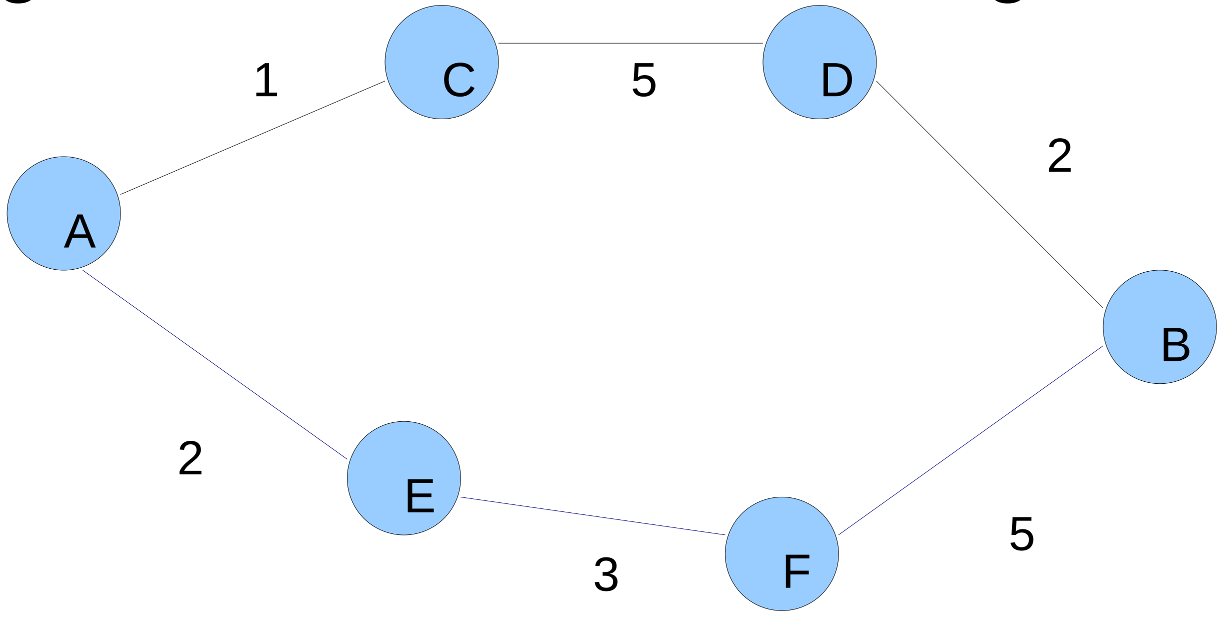
Pasos para desarrollar un algoritmo basado en PD

3) Calcular el valor de la solución optimal utilizando un enfoque ascendente.
   - Determinar el conjunto de subproblemas distintos a resolver (tamaño de la tabla)
   - Identificar los subproblemas con solucion trivial
   - Obtener los valores con un enfoque ascendente y almacenar los valores que vamos calculado en la tabla.
   - En etapas posteriores se utilizaran los valores previamente calculados

4) Determinar la solución óptima a partir de la información préviamente calculada.

Ejemplo de incumplimiento del POB

- Cálculo del camino simple (sin ciclos) más largo entre dos nodos de un grafo



El camino más largo de A a B es AEFB (10), pero el subcamino AEF (5) NO es el camino más largo de A a F (es ACDBF (13))

Ejemplo: Devolver cambio

- Compramos un artículo en una tienda
- el artículo cuesta 5.37 €
- pagamos con un billete de 10 €
- entonces nos tienen que devolver 4.63 €
- ¿Cómo nos devuelve el cambio con menor número de monedas?

Devolver cambio

El algoritmo Greedy era eficiente pero no eficaz:

- Si hay monedas de 1, 4 y 6 céntimos y hay que devolver 8 céntimos:
- el algoritmo greedy devuelve
  – 1 moneda de 6 céntimos
  – 2 monedas de 1 céntimo
- la solución óptima es 2 monedas de 4 céntimos
- Lo mejoramos con Programación Dinámica

Devolver cambio

El problema general es:

- monedas de n valores diferentes
- las monedas de tipo i tienen un valor de c_i>0 unidades
- hay un suministro ilimitado de monedas
- al cliente hay que devolverle un valor M
- hay que utilizar el menor número de monedas

Devolver cambio: POB?

- Se puede plantear la solución como una secuencia de decisiones x1,x2,...,xn: cuántas monedas de tipo 1, cuántas de tipo 2,...
- ¿Se cumple el POB?
- Sea m(i,j) el mínimo número de monedas para devolver una cantidad j usando solamente monedas de los tipos 1,2,...,i
- El problema original es m(n,M)

Demostración del POB

Si $x_1..x_n$ es óptima para m(n,M) entonces hay que demostrar que $x_1..x_{n-1}$ es óptima para m(n-1,M-x_n*c_n).

- Por contradicción
- Si no fuese así entonces existe $y_1..y_{n-1}$ tal que

  sum_1^{n-1} y_i < sum_1^{n-1} x_i y sum_1^{n-1} y_i*c_i = M-x_n*c_n
- Pero entonces, haciendo y_n=x_n tenemos que

  sum_1^{n} y_i*c_i = M, luego $y_1..y_n$ es una solución para m(n,M), y además

  sum_1^{n} y_i < sum_1^{n} x_i, luego $x_1..x_n$ no sería óptima

Definición recursiva de la solución óptima

- Si la solución para $m(i,j)$ no incluye una moneda de tipo i, entonces

  $m(i,j)=m(i-1,j)$
- Si la solución para $m(i,j)$ sí incluye una moneda de tipo i, entonces

  $m(i,j)=1+m(i,j-c_i)$
- Luego $m(i,j)=\min(\ m(i-1,j),1+m(i,j-c_i)\ )$

Definición recursiva

- Más exactamente
  - $m(i,j)=0$ si $j=0$
  - $m(i,j)=$infinito si $i=1$ y $1<=j<c_i$
  - $m(i,j)=1+m(i,j-c_i)$ si $i=1$ y $j>=c_i$
  - $m(i,j)=m(i-1,j)$ si $i>1$ y $j<c_i$
  - $m(i,j)=\min(\ m(i-1,j),1+m(i,j-c_i)\ )$ en otro caso
  - Necesitamos una tabla para almacenar los resultados de todos los subproblemas
  - Rellenamos la tabla de arriba a abajo y de izquierda a derecha

Devolver cambio: Algoritmo PD

```
For i=1 to n
   m[i,0]=0;
For i=1 to n
   For j=1 to M
      If (i==1 && j<c[i]) m[i,j]=10e30;
      Else if (i==1) m[i,j]=1+m[i,j-c[1]];
              Else if (j<c[i]) m[i,j]=m[i-1,j];
                      Else m[i,j]=min(m[i-1,j],1+m[i,j-c[i]]);
Return m[n,M];
```

La eficiencia del algoritmo es O(nM)

Ejemplo

$$m[i,j]=\min(\ m[i-1,j], 1+m[i,j-c[i]]\ )$$

$n= 3$, $M= 8$, $c= (1, 4, 6)$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
C_1= 1	0								
C_2 = 4	0								
C_3 = 6	0								

Casos base: m[i,0]=0

Ejemplo

m[i,j]=min(m[i-1,j],1+m[i,j-c[i]])

n= 3, M= 8, c= (1, 4, 6)

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
C_1= 1	0	1	2	3	4	5	6	7	8
C_2 = 4	0								
C_3 = 6	0								

m[1,j]=1+m[1,j-1]

Ejemplo

$$m[i,j]=\min(\ m[i-1,j],1+m[i,j-c[i]]\ )$$

n= 3, M= 8, c= (1, 4, 6)

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
C_1= 1	0	1	2	3	4	5	6	7	8
C_2 = 4	0	1	2	3	1	2	3	4	2
C_3 = 6	0								

$m[2,j]=\min(\ m[1,j],\ 1+m[2,j-4]\ )$

Ejemplo

m[i,j]=min(m[i-1,j],1+m[i,j-c[i]])

n= 3, M= 8, c= (1, 4, 6)

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
C_1= 1	0	1	2	3	4	5	6	7	8
C_2 = 4	0	1	2	3	1	2	3	4	2
C_3 = 6	0	1	2	3	1	2	1	2	2

m[3,j]=min(m[2,j], 1+m[3,j-6])

Ejemplo

$$m[i,j]=\min(m[i-1,j],1+m[i,j-c[i]])$$

n= 3, M= 8, c= (1, 4, 6)

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
C_1= 1	0	1	2	3	4	5	6	7	8
C_2 = 4	0	1	2	3	1	2	3	4	2
C_3 = 6	0	1	2	3	1	2	1	2	2

El valor de la solución óptima del problema original es m[3,8]=2 monedas

Esta tabla solo da el número mínimo de monedas necesario para cada valor de M.

¿Cómo calcular cuántas monedas de cada tipo hacen falta?

Cuántas monedas de cada tipo?

- Empezamos con m[n,M]: i=n; j=M;
- Si m[i,j] = m[i-1,j] no se escoge una moneda de tipo i y pasamos a estudiar m[i-1,j]: i=i-1;
- Si m[i,j] = 1+m[i,j-c[i]] se escoge una moneda de tipo i y pasamos a estudiar m[i,j-c[i]]: j=j-c[i]; monedas[i]++;
- Hasta que lleguemos a algún m[i,0], y ya no quede nada por pagar

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
C_1= 1	0	1	2	3	4	5	6	7	8
C_2 = 4	0	1	2	3	1	2	3	4	2
C_3 = 6	0	1	2	3	1	2	1	2	2

Ejercicio: adaptar el algoritmo de PD al caso en que hay un número limitado de monedas de cada tipo

Mochila 0/1

Tenemos un conjunto S de n objetos, donde cada objeto, i, tiene

- b_i – beneficio positivo
- w_i – peso positivo

- Objetivo: **Seleccionar los elementos que nos garantizan un beneficio máximo pero con un peso global menor o igual que W**
- Los objetos no se pueden partir

Mochila 0/1: POB?

- Se puede plantear la solución como una secuencia de decisiones x1,x2,...,xn: seleccionar o no el objeto 1, seleccionar o no el objeto 2,...
- ¿Se cumple el POB?
- Sea B(k,w) el mejor valor obtenido para un peso máximo de w usando solamente los objetos 1,2,...,k
- La solución del problema original es B(n,W)

Demostración del POB

Si $x_1..x_n$ es óptima para B(n,W) entonces hay que demostrar (por contradicción) que $x_1..x_{n-1}$ es óptima para:

- B(n-1,W) si x_n=0 ó B(n-1,W-w_n) si x_n=1
- Caso x_n=0. Si no fuese así entonces existe $y_1..y_{n-1}$ tal que

  sum_1^{n-1} y_i*b_i > sum_1^{n-1} x_i*b_i y sum_1^{n-1} y_i*w_i <=W
- Pero entonces, haciendo y_n=x_n=0 tenemos que

  sum_1^{n} y_i*w_i <= W, luego $y_1..y_n$ es una solución para B(n,W), y además

  sum_1^{n} y_i*b_i > sum_1^{n} x_i*b_i, luego $x_1..x_n$ no sería óptima

Demostración del POB

Caso x_n=1

- Si $x_1..x_{n-1}$ no es óptima para B(n-1,W-w_n) entonces existe $y_1..y_{n-1}$ tal que

  sum_1^{n-1} y_i*b_i > sum_1^{n-1} x_i*b_i y

  sum_1^{n-1} y_i*w_i <=W-w_n
- Pero entonces, haciendo y_n=x_n=1 tenemos que

  sum_1^{n-1} y_i*w_i+w_n <= W, luego $y_1..y_n$ es una solución para B(n,W), y además

  sum_1^{n-1} y_i*b_i +b_n> sum_1^{n-1} x_i*b_i +b_n, luego $x_1..x_n$ no sería óptima

Princ. de Optimalidad de Bellman

- La mejor selección de elementos del conjunto 1,2,...,k para una mochila de tamaño w se puede definir en función de selecciones de elementos de 1,2,...,k-1, para mochilas de tamaño menor.

- O bien

  es la ganancia para la mejor selección de elementos de 1,2,...,k-1 con peso máximo w (NO selecciono el objeto k)

- O bien

  es la ganancia de la mejor selección de elementos de 1,2,...,k-1 con peso máximo $w-w_k$ más la ganancia del elemento k, b_k. (SI selecciono el objeto k)

Definición Recursiva

$$B[k,w]=\begin{cases} B[k-1,w] & si \quad w_k > w \\ \max\left(B[k-1,w], B[k-1,w-w_k]+b_k\right) & en\ otro\ caso \end{cases}$$

Caso base: B[0,w] = 0, B[k,0] = 0

Podemos usar una tabla de tamaño (n+1)x(W+1) para almacenar los valores B[k,w]

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad j

	0	1	2	3	4	5
0						
1						
2						
3						
4						?

Problema Mochila: Ejemplo

Mochila de capacidad *W* = 5

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

Casos base: B(0,w)=B(k,0)=0

capacidad *j*

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					?

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

B(1,1)=B(0,1)=0

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad *j*

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

B(1,2)=max(B(0,2),12+B(0,2-2))=max(0,12+0)=12

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad *j*

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12			
2	0					
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

B(1,3)=max(B(0,3),12+B(0,3-2))=max(0,12+0)=12

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad *j*

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12		
2	0					
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

B(1,4)=max(B(0,4),12+B(0,4-2))=max(0,12+0)=12

capacidad *j*

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	
2	0					
3	0					
4	0					?

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

B(1,5)=max(B(0,5),12+B(0,5-2))=max(0,12+0)=12

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad *j*

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0					
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad *W* = 5

item	peso	ganancia
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

B(2,1)=max(B(1,1),10+B(1,1-1))=max(0,10+0)=10

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad *j*

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10				
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

B(2,2)=max(B(1,2),10+B(1,2-1))=max(12,10+0)=12

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad *j*

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12			
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad *W* = 5

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

B(2,3)=max(B(1,3),10+B(1,3-1))=max(12,10+12)=22

capacidad *j*

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22		
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

B(2,4)=max(B(1,4),10+B(1,4-1))=max(12,10+12)=22

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad j

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad *W* = 5

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

B(2,5)=max(B(1,5),10+B(1,5-1))=max(12,10+12)=22

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad *j*

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad *W* = 5

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

B(3,1)=B(2,1)=10

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad *j*

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10				
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

B(3,2)=B(2,2)=12

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad j

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12			
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

B(3,3)=max(B(2,3),20+B(2,3-3))=max(22,20+0)=22

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad *j*

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22		
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad W = 5

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

B(3,4)=max(B(2,4),20+B(2,4-3))=max(22,20+10)=30

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

capacidad j

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad *W* = 5

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

B(3,5)=max(B(2,5),20+B(2,5-3))=max(22,20+12)=32

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad *j*

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

B(4,1)=B(3,1)=10

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad *j*

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10				?

Problema Mochila: Ejemplo

Mochila de capacidad *W* = 5

item	peso	ganancia
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

B(4,2)=max(B(3,2),15+B(3,2-2))=max(12,15+0)=15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

capacidad *j*

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15			?

Problema Mochila: Ejemplo

Mochila de capacidad *W* = 5

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

B(4,3)=max(B(3,3),15+B(3,3-2))=max(22,15+10)=25

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad *j*

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25		?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

B(4,4)=max(B(3,4),15+B(3,4-2))=max(30,15+12)=30

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad *j*

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

B(4,5)=max(B(3,5),15+B(3,5-2))=max(32,15+22)=37

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad *j*

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	**37**

Algoritmo *01Knapsack*(*S, W*):

Input: Conjunto ***S de n*** objetos, beneficio ***bi*** y peso ***wi***; capacidad máxima ***W***

```
for w ← 0 to W do
    B[0,w] = 0
for k ← 1 to n do
    B[k,0] = 0
    for w ← 1 to wk -1 do
            B[k,w] = B[k-1,w]
    for w ← wk to W do
            if B[k-1,w −wk] + bk > B[k-1,w] then
                    B[k,w] = B[k-1,w −wk] + bk
            else B[k,w] = B[k-1,w]
return B[n,W]
```

Búsqueda de la Solución Optimal

- A partir de la matriz calculada por el algoritmo anterior

Si *B*[k,*w*] == ***B[k-1,w]***

entonces el objeto k no se selecciona y se pasa a estudiar el objeto k-1 para una mochila de capacidad w:

Problema B[k-1,w]

Si *B*[k,*w*] != ***B[k-1,w]***

se selecciona el objeto k, pasando a estudiar el objeto k-1 con una mochila de capacidad w-wk:

Problema B[k-1,w-wk]

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad j

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	**37**

Se seleccionan los objetos 4, 2 y 1

Multiplicación Encadenada de Matrices

- Dadas n matrices $A_1, A_2, \ldots, A_n$ con A_i de dimensión d_{i-1} x d_i
- Determinar el orden de multiplicación para minimizar el numero de multiplicaciones escalares.
- Suponemos que la multiplicación de una matriz p x e por otra e x r requiere per multiplicaciones escalares

$$C[i,j] = \sum_{k=1}^{e} A[i,k] * B[k,j]$$

Multiplicación encadenada de matrices

- Un producto de matrices se dice que está completamente parentizado si está constituido por una sola matriz, o por el producto completamente parentizado de dos matrices, cerrado por paréntesis.
- La multiplicación de matrices es asociativa, y por tanto todas las parentizaciones producen el mismo resultado.
- Pero el número de cálculos (multiplicaciones escalares) puede variar de una parentización a otra.

Multiplicación Encadenada de Matrices

- ¿Por qué es importante la parentización?

- Ejemplo
  B es 3 × 100
  C es 100 × 5
  D es 5 × 5

  (B*C)*D necesita 1500 + 75 = 1575 operaciones
  B*(C*D) necesita 1500 + 2500 = 4000 operaciones

Multiplicación Encadenada de Matrices

- El producto $A_1 A_2 A_3 A_4$ puede parentizarse completamente de 5 formas distintas
  - $(A_1 (A_2(A_3A_4)))$
  - $(A_1 ((A_2A_3)A_4))$
  - $((A_1 A_2)(A_3A_4))$
  - $((A_1 (A_2A_3))A_4)$
  - $(((A_1 A_2)A_3)A_4)$

A = **A** x **A**$_2$ x **A** x **A** x

Multiplicación Encadenada de Matrices

Orden 1 (A **x (A x (A x A)))**

Costo(A x A) = 50 x 1 x 100

Costo(A x (A x A)) = 20 x 50 x 100

Costo(A x (A x (A x A))) = 10 x 20 x 100

Costo total* = *125000 multiplicaciones

Orden 4 ((A **x (A x A)) x A)**

Costo(A x A) = 20 x 50 x 1

Costo(A x (A x A)) = 10 x 20 x 1

Costo((A x (A x A)) x A) = 10 x 1 x 100

Costo total* = *2200 multiplicaciones

Recuento del número de parentizaciones

La enumeración de todas las parentizaciones posibles no proporciona un método eficiente.
Notemos el número de parentizaciones de una sucesión de n matrices por $P(n)$.
Como podemos dividir una sucesión de n matrices en dos (las k primeras y las n-k siguientes) para cualquier $k = 1,2,\ldots,n-1$, y entonces parentizar las dos subsucesiones resultantes independientemente, obtenemos la recurrencia:

$$
\begin{aligned}
P(n) &= 1 && \text{si } n = 1 \\
&= \textstyle\sum_{k=1..n-1} P(k)\times P(n-k) && \text{si } n \geq 2
\end{aligned}
$$

Recuento del número de parentizaciones

La solución de esa ecuación es la sucesión de los Números de Catalan (que también cuenta el número de árboles binarios con n+1 hojas)

$$P(n) = C(n-1)$$

Donde

$$C(n) = (n+1)^{-1} C_{2n,n} = \frac{\binom{2n}{n}}{n+1}$$

es de orden exponencial, $4^n/n^{3/2}$,

Por tanto el método de la fuerza bruta es una pobre estrategia para determinar la parentización optimal de una cadena de matrices.

Multiplic. Encadenada de Matrices

- Se puede plantear mediante PD? Se cumple el POB?
- La secuencia de decisiones es dónde se coloca el primer paréntesis, dónde el segundo,...
  - La solución óptima se puede definir en términos de soluciones óptimas a problemas de tamaño menor.
  - Necesariamente tiene que haber una multiplicación final (la de mayor nivel) en el cálculo de la solución óptima.
  - Supongamos que la última multiplicación se realiza en la posición i:

$$(A_1 * \ldots * A_i) * (A_{i+1} * \ldots * A_n).$$

  - Si hubiera una solución mejor para algún subproblema, podríamos usarla en lugar de la anterior y obtendríamos una solución mejor que la óptima. Contradicción.

Multiplic. Encadenada de Matrices

- Sea $p_{k-1}xp_k$ la dimensión de la matriz A_k
- Problema: Multiplicar $(A_1A_2...A_kA_{k+1}A_{k+2}\dots A_n)$
- Supongamos que parentizamos en k
  - $(A_1A_2...A_k)$ $(A_{k+1}A_{k+2}\dots A_n)$

- Si llamamos N[i,j] al número de operaciones necesarias para multiplicar $A_i*A_{i+1}*\dots*A_j$.

$$\mathbf{N[1,n] = N[1,k]+N[k+1,n]+p_0p_kp_n.}$$

Definición Recursiva Solución Optimal:

- Si i=j entonces

  N[i,j] = 0

- Para 1 <= i < j <= n

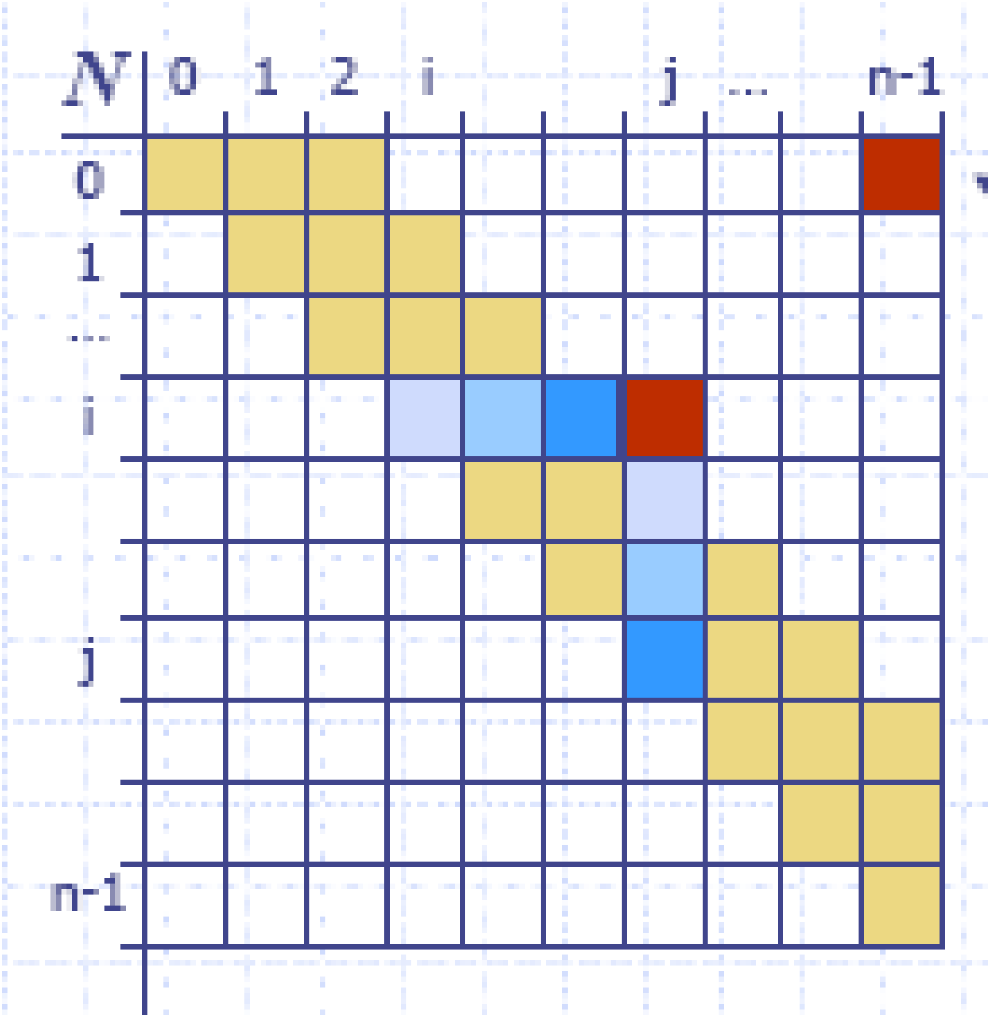
  $N[i,j]= \min\{i<=k<j\} (N[i,k]+N[k+1,j]+p_{i-1}p_kp_j)$

A partir de la definición recursiva se puede ver como se produce el solapamiento de los subproblemas

Primero se solucionan los suproblemas triviales (tamaño 0) para en cada paso ir resolviendo subproblemas de tamaño 1,2,3,…

Tenemos $O(n^2)$ subproblemas distintos

Para calcular N[i,j] necesitamos los valores almacenados en la fila i columna j



Multiplicación encadenada de matrices

- **Tablas usadas por el algoritmo.**
  - Sea **N** una matriz [1..n, 1..n] de enteros. El algoritmo usará la mitad de la matriz.

	j= 1	2	3	4	
i=1	0	X	X	X	3
2		0	X	X	2
3			0	X	1
4				0	0

- **Forma de rellenar la tabla.**
  - Inicializar la matriz. Para todo i, desde 1 hasta n. N[i, i] = 0
  - Aplicar la ecuación de recurrencia por diagonales.

$$N[i, j] = \min_{i \leq k < j} (N[i, k] + N[k+1, j] + p_{i-1}p_kp_j)$$

- **Ejemplo**. n= 4, p = (p_0=10, p_1=20, p_2=50, p_3=1, p_4=100)

	j= 1	2	3	4
i=1	0	10.000	1.200	2.200
2		0	1.000	3.000
3			0	5.000
4				0

Multiplicación encadenada de matrices

- **Tablas usadas por el algoritmo.**
  - Sea **N** una matriz [1..n, 1..n] de enteros. El algoritmo usará la mitad de la matriz.

	j= 1	2	3	4	
i=1	0	X	X	X	3
2		0	X	X	2
3			0	X	1
4				0	0

- **Forma de rellenar la tabla.**
  - Inicializar la matriz. Para todo i, desde 1 hasta n. N[i, i] = 0
  - Aplicar la ecuación de recurrencia por diagonales.

$$N[i, j] = \min_{i \leq k < j} (N[i, k] + N[k+1, j] + p_{i-1}p_kp_j)$$

- **Ejemplo**. n= 4, p = (p_0=10, p_1=20, p_2=50, p_3=1, p_4=100)

	j= 1	2	3	4
i=1	0	10.000	1.200	2.200
2		0	1.000	3.000
3			0	5.000
4				0

$N[1,2]=N[1,1]+N[2,2]+p_0p_1p_2=0+0+10000=10000$

Multiplicación encadenada de matrices

- **Tablas usadas por el algoritmo.**
  - Sea **N** una matriz [1..n, 1..n] de enteros. El algoritmo usará la mitad de la matriz.

	j= 1	2	3	4	
i=1	0	X	X	X	3
2		0	X	X	2
3			0	X	1
4				0	0

- **Forma de rellenar la tabla.**
  - Inicializar la matriz. Para todo i, desde 1 hasta n. N[i, i] = 0
  - Aplicar la ecuación de recurrencia por diagonales.

$$N[i, j] = \min_{i \leq k < j} (N[i, k] + N[k+1, j] + p_{i-1}p_kp_j)$$

- **Ejemplo**. n= 4, p = (p_0=10, p_1=20, p_2=50, p_3=1, p_4=100)

	j= 1	2	3	4
i=1	0	10.000	1.200	2.200
2		0	1.000	3.000
3			0	5.000
4				0

$N[2,3]=N[2,2]+N[3,3]+p_1p_2p_3=0+0+1000=1000$

Multiplicación encadenada de matrices

- **Tablas usadas por el algoritmo.**
  - Sea **N** una matriz [1..n, 1..n] de enteros. El algoritmo usará la mitad de la matriz.

	j= 1	2	3	4	
i=1	0	X	X	X	3
2		0	X	X	2
3			0	X	1
4				0	0

- **Forma de rellenar la tabla.**
  - Inicializar la matriz. Para todo i, desde 1 hasta n. N[i, i] = 0
  - Aplicar la ecuación de recurrencia por diagonales.

$$N[i, j] = \min_{i \leq k < j} (N[i, k] + N[k+1, j] + p_{i-1}p_k p_j)$$

- **Ejemplo**. n= 4, p = (p_0=10, p_1=20, p_2=50, p_3=1, p_4=100)

	j= 1	2	3	4
i=1	0	10.000	1.200	2.200
2		0	1.000	3.000
3			0	5.000
4				0

$N[3,4]=N[3,3]+N[4,4]+p_2p_3p_4=0+0+5000=5000$

Multiplicación encadenada de matrices

- # Tablas usadas por el algoritmo.
  - Sea **N** una matriz [1..n, 1..n] de enteros. El algoritmo usará la mitad de la matriz.

	j= 1	2	3	4	
i=1	0	X	X	X	3
2		0	X	X	2
3			0	X	1
4				0	0

- **Forma de rellenar la tabla.**
  - Inicializar la matriz. Para todo i, desde 1 hasta n. N[i, i] = 0
  - Aplicar la ecuación de recurrencia por diagonales.

$$N[i, j] = \min_{i \leq k < j} (N[i, k] + N[k+1, j] + p_{i-1}p_kp_j)$$

- **Ejemplo**. n= 4, p = (p_0=10, p_1=20, p_2=50, p_3=1, p_4=100)

	j= 1	2	3	4
i=1	0	10.000	1.200	2.200
2		0	1.000	3.000
3			0	5.000
4				0

$N[1,3]=\min(N[1,1]+N[2,3]+p_0p_1p_3, N[1,2]+N[3,3]+p_0p_2p_3)=\min(0+1000+200, 10000+0+500)=1200$

Multiplicación encadenada de matrices

- **Tablas usadas por el algoritmo.**
  - Sea **N** una matriz [1..n, 1..n] de enteros. El algoritmo usará la mitad de la matriz.

	j= 1	2	3	4	
i=1	0	X	X	X	3
2		0	X	X	2
3			0	X	1
4				0	0

- **Forma de rellenar la tabla.**
  - Inicializar la matriz. Para todo i, desde 1 hasta n. N[i, i] = 0
  - Aplicar la ecuación de recurrencia por diagonales.

$$N[i, j] = \min_{i \leq k < j} (N[i, k] + N[k+1, j] + p_{i-1}p_k p_j)$$

- **Ejemplo**. n= 4, p = (p_0=10, p_1=20, p_2=50, p_3=1, p_4=100)

	j= 1	2	3	4
i=1	0	10.000	1.200	2.200
2		0	1.000	3.000
3			0	5.000
4				0

$N[2,4]=\min(N[2,2]+N[3,4]+p_1p_2p_4,\ N[2,3]+N[4,4]+p_1p_3p_4)=\min(0+5000+100000, 1000+0+2000)=3000$

Multiplicación encadenada de matrices

- **Tablas usadas por el algoritmo.**
  - Sea **N** una matriz [1..n, 1..n] de enteros. El algoritmo usará la mitad de la matriz.

	j= 1	2	3	4	
i=1	0	X	X	X	3
2		0	X	X	2
3			0	X	1
4				0	0

- **Forma de rellenar la tabla.**
  - Inicializar la matriz. Para todo i, desde 1 hasta n. N[i, i] = 0
  - Aplicar la ecuación de recurrencia por diagonales.

$$N[i, j] = \min_{i \le k < j} (N[i, k] + N[k+1, j] + p_{i-1}p_kp_j)$$

- **Ejemplo**. n= 4, p = (p_0=10, p_1=20, p_2=50, p_3=1, p_4=100)

	j= 1	2	3	4
i=1	0	10.000	1.200	2.200
2		0	1.000	3.000
3			0	5.000
4				0

$N[1,4]=\min(N[1,1]+N[2,4]+p_0p_1p_4,\ N[1,2]+N[3,4]+p_0p_2p_4,\ N[1,3]+N[4,4]+p_0p_3p_4)=2200$

Multiplicación encadenada de matrices

- ¿Cuál es el orden de complejidad de este algoritmo?

  Para calcular cada valor necesitamos O(n)

  El algoritmo final es de orden $O(n^3)$

- En la posición **N[1, n]** tenemos almacenado el número mínimo de multiplicaciones escalares necesario (para la ordenación que es óptima). Necesitamos calcular cuál es esta ordenación óptima.
- Usar una matriz auxiliar **Mejork [1..n, 1..n]** en la que se almacene el índice donde se alcanzó el mínimo (mejor valor de **k)** para cada subproblema.
- En el **ejemplo anterior**.

Mejork	j= 1	2	3	4
i=1	-	1	1	3
2		-	2	3
3			-	3
4				-

Cálculo de la solución

Mejork	j= 1	2	3	4
i=1	-	1	1	3
2		-	2	3
3			-	3
4				-

Mejork[1,4]=3, luego los subproblemas son $A_{1..3}$ y $A_{4..4}$ o sea $(A_1A_2A_3)A_4$
Mejork[1,3]=1, luego tenemos $A_{1..1}$ y $A_{2..3}$ o sea $A_1(A_2A_3)$

Luego la parentización óptima es $((A_1(A_2A_3))A_4)$

Algoritmo

- MultCadenaMatrices(n)
  - for i= 1 to n

    N[i,i] = 0
  - for l = 2 to n

    for i = 1 to n-l+1

        j=i+l-1

        N[i,j] = inf.

        for k=i to j-1

            q=N[i,k] + N[k+1,j] + p[i-1]p[k]p[j]

            if q < N[i,j]

                N[i,j] = q

                Mejork[i,j] = k

- MultplCadMatrices(A, s, i, j)

  if j>i

  ```
  x = MultplCadMatrices (A, s, i, s[i,j])
  y = MultplCadMatrices (A, s, s[i,j]+1, j)
  return MultMatrices(x, y)
  ```

  else return Ai

  s representa a Mejork y MultMatrices es la multiplicación estándar de 2 matrices.

Subsecuencia Común de Mayor Longitud (LCS)

Dadas dos secuencias de símbolos X e Y, ¿cuál es la subsecuencia común a X e Y de longitud mayor?

Ej: X= {A B C B D A B }, Y= {B D C A B A}

Subsec. Común de Mayor longitud :

X = A **B** **C** **B** D **A** B

Y = **B** D **C** A **B** **A**

También puede ser BDAB

Subsecuencia Común de Mayor Longitud (LCS)

Aplicaciones en bioinformática (genómica) y en comparación de ficheros (diff).

Un algoritmo de fuerza bruta compararía cualquier subsecuencia de X con los símbolos de Y.

Si $|X| = m$, $|Y| = n$, hay que contrastar 2^m subsecuencias de X contra los n elementos de Y.

Eso daría un algoritmo de orden $O(n2^m)$

LCS: POB?

- Sin embargo, LCS tiene *subestructuras optimales*: las soluciones a los subproblemas son parte de la solución final.

X = A **B** **C** **B** D **A** B

Y = **B** D **C** A **B** **A**

Si BCBA es solución optimal => BCB debe ser solución optimal, para qué subproblemas?

Para X' = ABCBD (quitamos AB a X por la dcha.) y

Y' = BDCAB (quitamos A a Y por la dcha.)

Si BCB no es optimal para X' e Y' entonces BCBA no puede ser optimal para X e Y: Existe una secuencia optimal “UUUU” para X’ e Y’ de longitud mayor que 3, entonces “UUUUA” es una secuencia común de longitud mayor que 4 para X e Y, y entonces BCBA no es óptima.

LCS: subproblemas

- Subproblemas: "Encontrar LCS para pares de prefijos de X e Y"

- Definimos X_i, Y_j los prefijos de X e Y de longitud *i* y *j* respectivamente

$$X = \langle x_1, x_2, \ldots, x_m \rangle,\ Y = \langle y_1, y_2, \ldots, y_n \rangle$$

$$X_i = \langle x_1, x_2, \ldots, x_i \rangle,\ Y_j = \langle y_1, y_2, \ldots, y_j \rangle$$

- Definimos *c[i,j]* la longitud de LCS para X_i e Y_j
- Entonces, LCS de X e Y será *c[m,n]*

LCS

Supongamos que la sub-secuencia común de mayor longitud LCS de X e Y es

$$Z_k = \langle z_1, z_2, \ldots , z_k \rangle$$

Si $x_m = y_n$ entonces $z_k = x_m = y_n$ y Z_{k-1} es una LCS de X_{m-1} e Y_{n-1}

En otro caso o bien Z_k es una LCS de X_{m-1} e Y_n o una LCS de X_m e Y_{n-1} (si $z_k = x_m$ entonces y_n no puede estar en LCS, de ahí que Z_k sea LCS de X_m e Y_{n-1}; si $z_k != x_m$, de ahí que Z_k sea LCS de X_{m-1} e Y_n)

$$c[i, j] = \begin{cases} c[i-1, j-1]+1 & \text{si } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{en otro caso} \end{cases}$$

LCS: Definición Recursiva

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{si } x[i]=y[j], \\ \max(c[i,j-1],c[i-1,j]) & \textit{encasocontrario} \end{cases}$$

- *Inicio: i = j = 0* (subcadena vacía de x e y)

  (*c[0,0] = 0*)

- LCS de la cadena vacía y cualquier otra cadena es vacía, por tanto para cada par i,j:

  c[0, j] = c[i,0] = 0

Algoritmo LCS

LCS-Length(X, Y)

1. m = length(X) // get the # of symbols in X
2. n = length(Y) // get the # of symbols in Y
3. for i = 1 to m c[i,0] = 0 // special case: Y_0
4. for j = 1 to n c[0,j] = 0 // special case: X_0
5. for i = 1 to m // for all X_i
6. for j = 1 to n // for all Y_j
7. if (x[i] == y[j])
8. c[i,j] = c[i-1,j-1] + 1
9. else c[i,j] = max(c[i-1,j], c[i,j-1])
10. return c

Eficiencia O(nm)

Ejemplo LCS

- X = ABCB
- Y = BDCAB

LCS(X, Y) = BCB

X = A **B** **C** **B**

Y = **B** D **C** A **B**

Ejemplo LCS (0)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5

Ejemplo LCS (1)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

for i = 1 to m $\qquad$ c[i,0] = 0
for j = 1 to n $\qquad$ c[0,j] = 0

Ejemplo LCS (2)

ABCB
BDCAB

i		j	0	1	2	3	4	5
		Yj		**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**				
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

Ejemplo LCS (3)

ABCB
BDCAB

j		0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**		
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

Ejemplo LCS (4)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

Ejemplo LCS (5)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1** →	**1**
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

Ejemplo LCS (6)

ABCB
BDCAB

i		j	0	1	2	3	4	5
		Yj		**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**				
3	**C**		**0**					
4	**B**		**0**					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

Ejemplo LCS (7)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	
3	**C**		**0**					
4	**B**		**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

Ejemplo LCS (8)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**					
4	**B**		**0**					

if (X_i == Y_j)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

Ejemplo LCS (10)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**			
4	**B**	**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

Ejemplo LCS (11)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**		
4	**B**	**0**					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

Ejemplo LCS (12)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**					

if ($X_i == Y_j$)
 c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

Ejemplo LCS (13)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**	**1**				

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

Ejemplo LCS (14)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

Ejemplo LCS (15)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**	**1**	**1**	**2**	**2**	3

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

Cómo encontrar la subsecuencia LCS

Cada *c[i,j]* depende de *c[i-1,j] y c[i,j-1]*

O bien de *c[i-1, j-1]*

Por tanto, a partir del valor c[i,j] podremos averiguar cómo se determinó

2	2
2	3

Por ejemplo

c[i,j] = c[i-1,j-1] +1 = 2+1=3

Cómo encontrar la subsecuencia LCS

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i]=y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

- Empezamos desde *c[m,n]* y vamos hacia atrás
- Si *c[i,j] = c[i-1, j-1]+1* y *x[i]=y[j]*, guardamos *x[i]* (porque *x[i]* pertenece a LCS): i=i-1; j=j-1
- Si *c[i,j] = c[i, j-1]* : j=j-1
- Si *c[i,j] = c[i-1, j]* : i=i-1
- Si i=0 ó j=0 (alcanzamos el principio), devolvemos los caracteres almacenados en orden inverso.

Encontramos LCS

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

Encontramos LCS

		j	0	1	2	3	4	5
i			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	← **1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	← **2**	**2**
4	**B**		**0**	**1**	**1**	**2**	**2**	**3**

LCS : **B C B**

Caminos mínimos: Algoritmo de Floyd

Problema:
Calcular el camino más corto que une cada par de vértices de un grafo, considerando que no hay pesos negativos.

Posibles soluciones:

- Por fuerza bruta (de orden exponencial).
- Aplicar el algoritmo de Dijkstra para cada vértice.
- Algoritmo de Floyd (programación dinámica).

Caminos mínimos: POB?

- Camino mínimo entre dos vértices i y j
- Se puede plantear mediante PD? Se cumple el POB?
- La secuencia de decisiones es cuál es el primer vértice del camino, cuál el segundo,...
- Si el camino mínimo de i a j pasa por k, entonces los caminos de i a k y de k a j son también mínimos (si no lo fueran encontraríamos un camino de i a j mejor que el mínimo)
- Luego de cumple el principio de optimalidad de Bellman

Definición recursiva

$D_k(i,j)$: valor del camino más corto de i a j usando sólo los k primeros vértices del grafo, {1,2,...,k} como nodos intermedios.

Si el camino de i a j usando sólo los k primeros vértices NO pasa por k:

$$\mathbf{D_k(i,j)=D_{k-1}(i,j)}$$

Si el camino de i a j usando sólo los k primeros vértices SÍ pasa por k:

$$\mathbf{D_k(i,j)=D_{k-1}(i,k)+D_{k-1}(k,j)}$$

Definición recursiva

D_k(i,j): valor del camino más corto de i a j usando sólo los k primeros vértices del grafo, {1,2,...,k} como nodos intermedios.

Expresión recursiva:

$$D_k(i,j) = \min\{D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j)\}$$

Caso base:

$$D_0(i,j) = c_{ij}$$

Matriz de adyacencia, con c_{ij}=infinito si no hay arco de i a j

La solución del problema original el D_n(i,j)

Algoritmo de Floyd (1962)

```
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        D[i][j] = coste(i,j);

for (k=1; k<=n; k++)
    for (i=1; i<=n; i++)
        for (j=1; j<=n; j++)
            if (D[i][k] + D[k][j] < D[i][j] )
               D[i][j] = D[i][k] + D[k][j];
```

En la k-esima iteración los valores de la fila k y la columna k no cambian pues D[k,k]=0. Por eso se puede utilizar una única matriz D en vez de una para D_k y otra para D_{k-1}

Orden de eficiencia $O(n^3)$

$$D_k(i,j) = \min\{D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j)\}$$

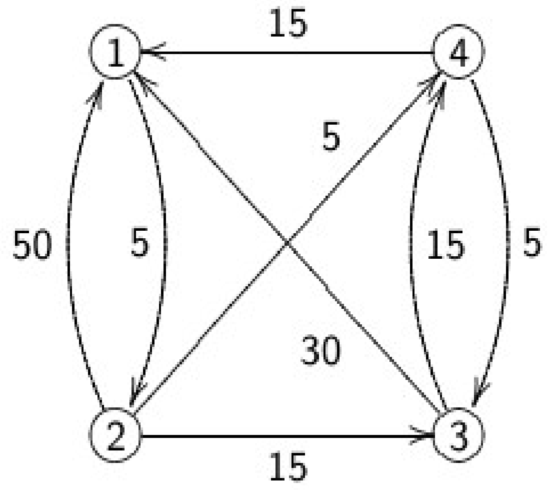
Floyd: cálculo de la solución

- Si además de conocer el valor del camino mínimo queremos conocer el camino en sí, empleamos otra matriz P, y el bucle interno del algoritmo sería:

```
if (D[i][k] + D[k][j] < D[i][j] )
                 D[i][j] = D[i][k] + D[k][j];
                 P[i][j] = k;
```

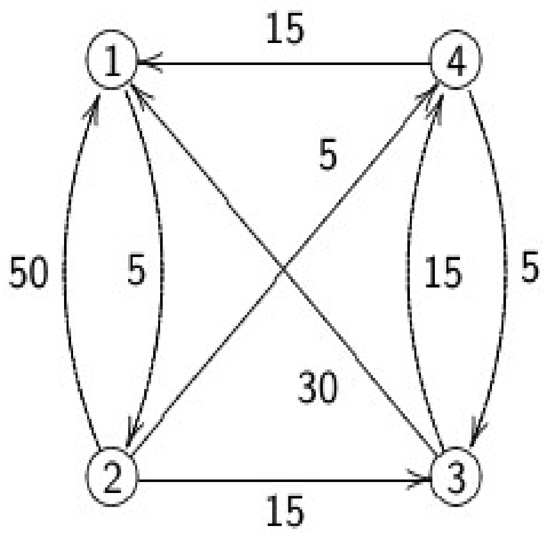
- Si al terminar, P[i][j]=0 el camino es el directo de i a j
- Si P[i][j]=k, entonces el camino pasa por k. Se analizan P[i][k] y P[k][j]

Floyd: ejemplo



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

Floyd: ejemplo, k=1



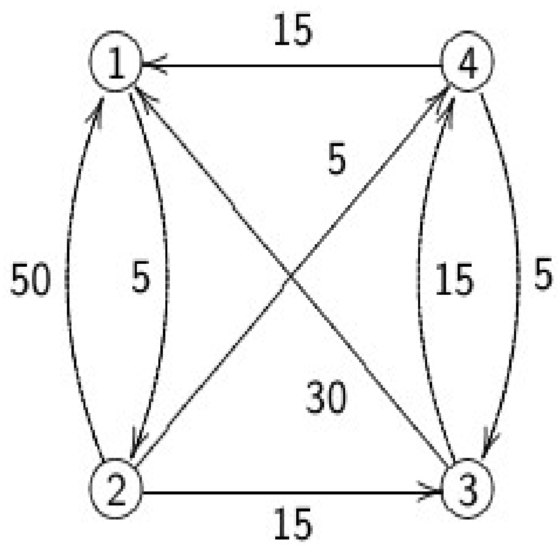
D(3,2)=inf, D(3,1)+D(1,2)=35, P(3,2)=1

D(4,2)=inf, D(4,1)+D(1,2)=20, P(4,2)=1

D(4,3)=5, D(4,1)+D(1,3)=inf, P(4,3)=0

$$D_0 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix} \qquad D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Floyd: ejemplo, k=2



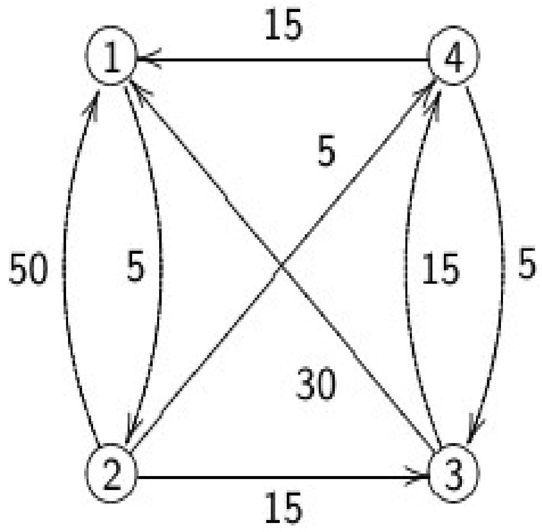
D(1,3)=inf, D(1,2)+D(2,3)=20, P(1,3)=2

D(1,4)=inf, D(1,2)+D(2,4)=10, P(1,4)=2

D(4,3)=5, D(4,2)+D(2,3)=35, P(4,3)=0

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

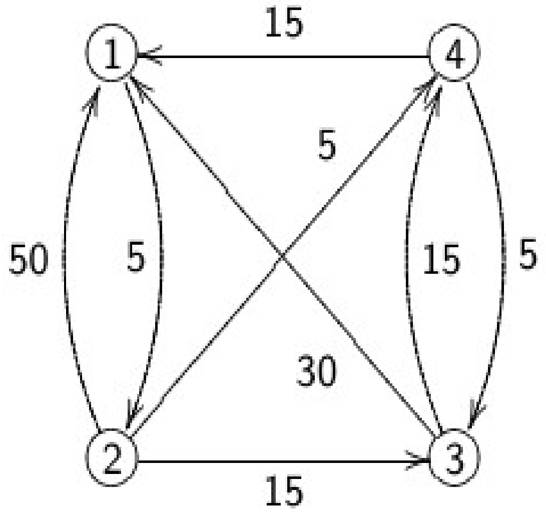
Floyd: ejemplo, k=3



D(2,1)=50, D(2,3)+D(3,1)=45, P(2,1)=3

$$D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Floyd: ejemplo, k=4



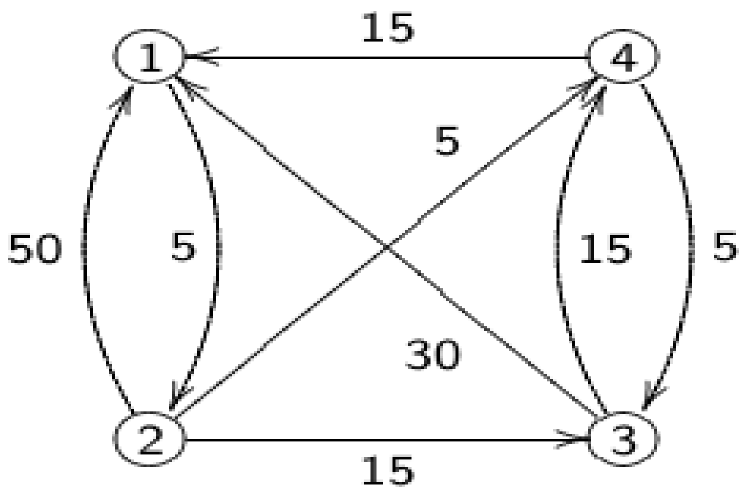
D(1,3)=20, D(1,4)+D(4,3)=15, P(1,3)=4

D(2,1)=45, D(2,4)+D(4,1)=20, P(2,1)=4

D(2,3)=15, D(2,4)+D(4,3)=10, P(2,3)=4

$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Floyd: cálculo de la solución



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

Ejemplo:

Camino de 1 a 3: P(1,3)=4

Camino de 1 a 4 y de 4 a 3
P(1,4)=2 P(4,3)=0
Camino de 1 a 2 y de 2 a 4
P(1,2)=0 P(2,4)=0

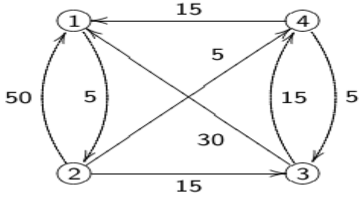
El camino mínimo de 1 a 3 es
1-2-4-3

Floyd: cálculo de la solución

Ejemplo:
Camino de 1 a 3: P(1,3)=4

Camino de 1 a 4 y de 4 a 3
P(1,4)=2   P(4,3)=0
Camino de 1 a 2 y de 2 a 4
P(1,2)=0   P(2,4)=0

El camino mínimo de 1 a 3 es
1-2-4-3



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

```
CalculaCamino(i,j)

 if (P(i,j)==0)
    print i-j
else
    CalculaCamino(i,P(i,j))
    CalculaCamino(P(i,j),j)
```

CalculaCamino(1,3)

1-2 2-4 4-3

Distancia de edición

También conocida como distancia Levenshtein, mide la diferencia entre dos cadenas s y t como el número mínimo de operaciones de edición (eliminar un carácter, añadir un carácter o cambiar un carácter) que hay que realizar para convertir una cadena en otra:

d(“data minin”, “data mining”) = 1 (añadir)

d(“defecto”, “efecto”) = 1 (eliminar)

d(“poda”, “boda”) = 1 (cambiar)

d(“night”,”noche”) = d(“natch”,”noche”) = 3

Aplicaciones: Correctores ortográficos, reconocimiento de voz, detección de plagios, análisis de ADN…

Distancia de edición

$$d(i,j)=\begin{cases} d(i-1,j-1) & si \quad s[i]=t[j] \\ 1+\min\{\underset{\text{Eliminar}}{d(i-1,j)},\underset{\text{Añadir}}{d(i,j-1)},\underset{\text{Sustituir}}{d(i-1,j-1)}\} & si \quad s[i]\neq t[j] \end{cases}$$

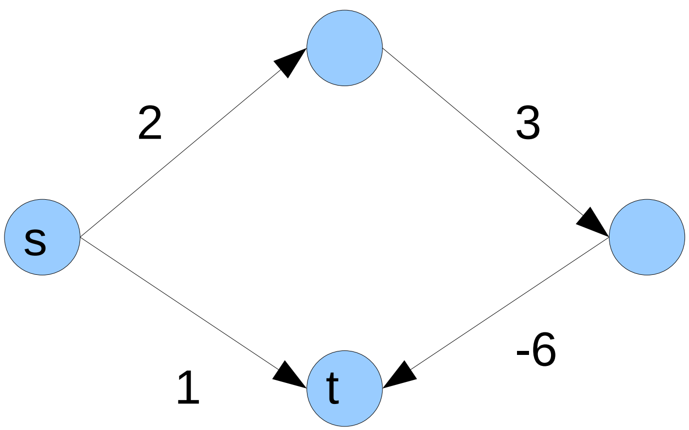
```
int LevenshteinDistance (string s[1..m], string t[1..n])
{
  for (i=0; i<=m; i++) d[i,0]=i;
  for (j=0; j<=n; j++) d[0,j]=j;

  for (i=1; i<=m; i++)
    for (j=1; j<=n; j++)
      if (s[i]==t[j])
         d[i,j] = d[i-1, j-1]
      else
         d[i,j] = 1+ min(d[i-1,j],d[i,j-1],d[i-1,j-1]);

  return d[m,n];
}
```

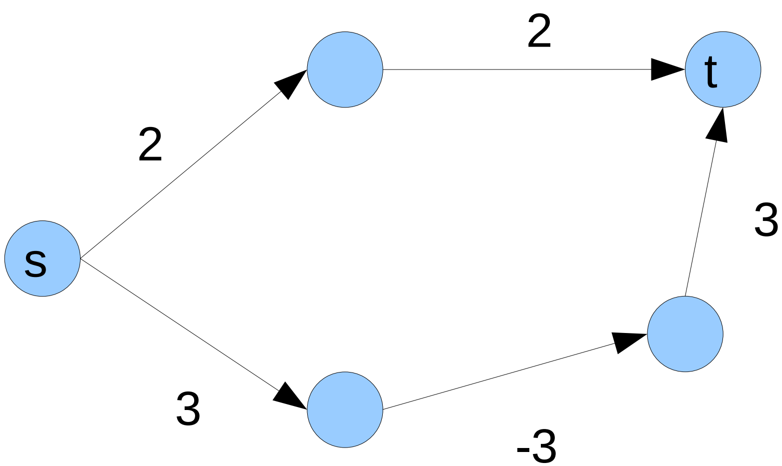
Caminos mínimos: Arcos con peso negativo

Si tenemos arcos con pesos negativos, no podemos usar ni el algoritmo de Dijkstra ni el de Floyd para calcular caminos mínimos.



Caminos mínimos: Arcos con peso negativo

- Tampoco podemos sumar a todos los arcos una cantidad para hacerlos positivos, eso altera los caminos mínimos



Algoritmo de Bellman-Ford

Podemos utilizar el algoritmo de Bellman-Ford, basado en programación dinámica y de orden **O(EV)**, siempre y cuando no tengamos ciclos de peso negativo.

Si existe algún ciclo de peso negativo, entonces no existe un camino mínimo entre dos nodos (pasando varias veces por el ciclo se va reduciendo el costo).

Si un grafo no tiene ciclos negativos, entonces hay un camino mínimo de s a t que es simple (no repite nodos) y tiene a lo sumo n-1 arcos

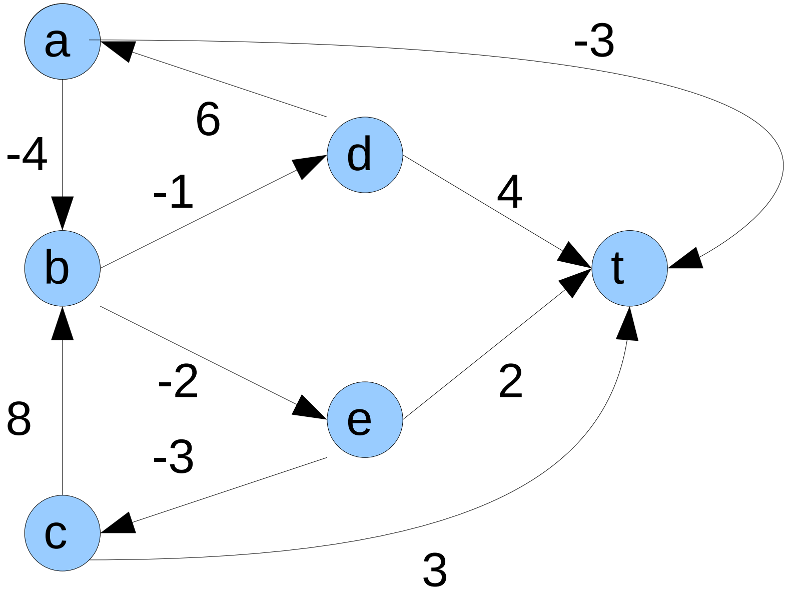
Planteamiento

- El objetivo es calcular el camino mínimo desde el nodo s hasta el nodo t
- Sea $D_i(v)$ el costo del camino mínimo entre el nodo v y el nodo t usando como mucho i arcos.
- El problema original es calcular $D_{n-1}(s)$

Ecuación recursiva

- Sea P el camino mínimo de v a t con costo $D_i(v)$.
- Si P usa como mucho i-1 arcos entonces $D_i(v)=D_{i-1}(v)$
- Si P usa i arcos y el primer arco es v->w entonces $D_i(v)=c_{vw}+D_{i-1}(w)$
- $D_i(v)=\min(D_{i-1}(v), \min_w(c_{vw}+D_{i-1}(w)))$
- $D_o(t)=0$
- $D_0(w)= \infty$, w!=t

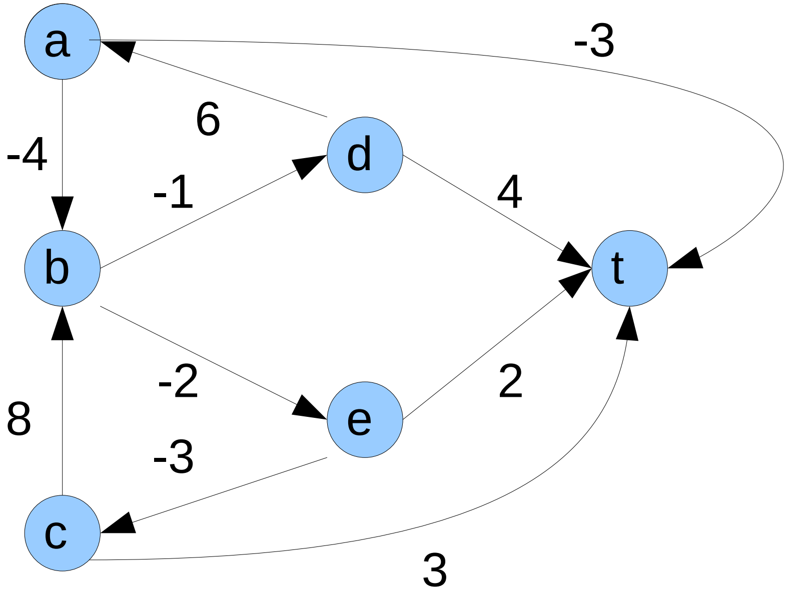
Ejemplo



	0	1	2	3	4	5
t	0					
a	inf					
b	inf					
c	inf					
d	inf					
e	inf					

$$D_i(v)=\min(D_{i-1}(v), \min_w(c_{vw}+D_{i-1}(w)))$$

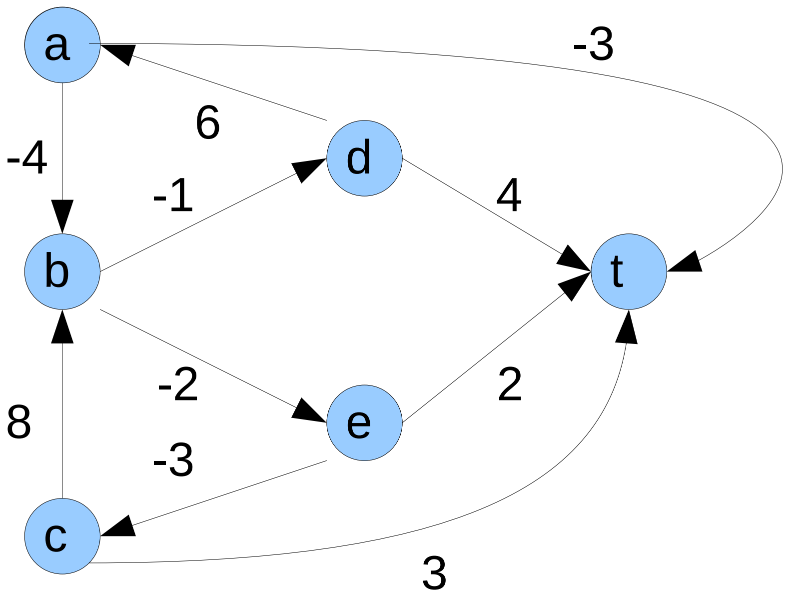
Ejemplo



	0	1	2	3	4	5
t	0	0				
a	inf	-3				
b	inf	inf				
c	inf	3				
d	inf	4				
e	inf	2				

$$D_i(v)=\min(D_{i-1}(v), \min_w(c_{vw}+D_{i-1}(w)))$$

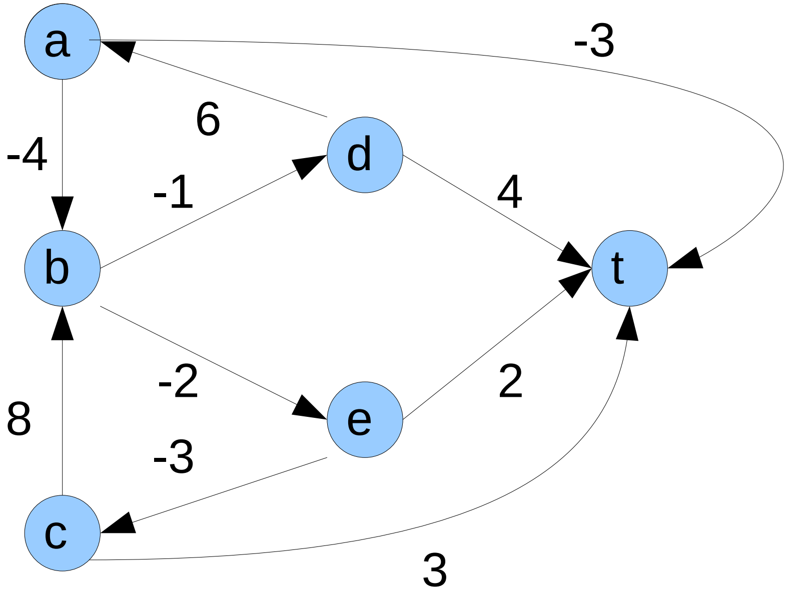
Ejemplo



	0	1	2	3	4	5
t	0	0	0			
a	inf	-3	-3			
b	inf	inf	0			
c	inf	3	3			
d	inf	4	3			
e	inf	2	0			

$$D_i(v)=\min(\ D_{i-1}(v),\ \min_w(c_{vw}+D_{i-1}(w)))$$

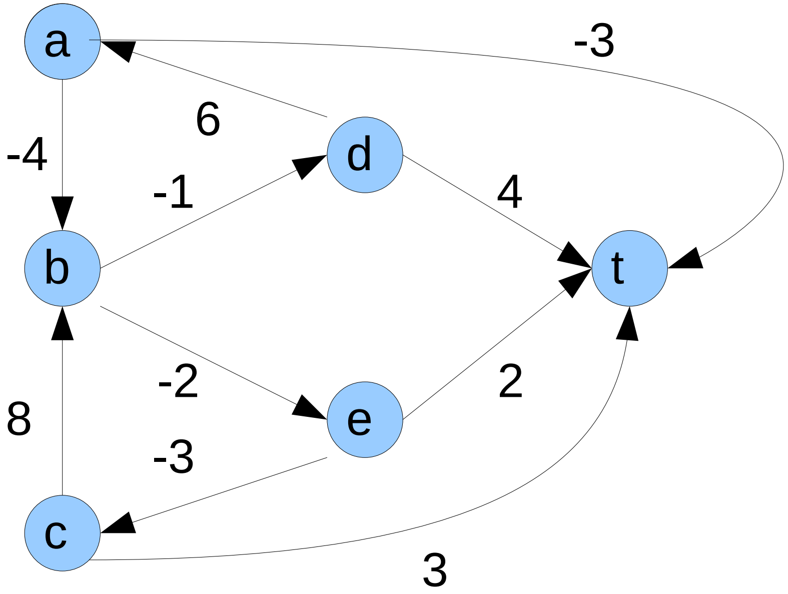
Ejemplo



	0	1	2	3	4	5
t	0	0	0	0		
a	inf	-3	-3	-4		
b	inf	inf	0	-2		
c	inf	3	3	3		
d	inf	4	3	3		
e	inf	2	0	0		

$$D_i(v)=\min(D_{i-1}(v), \min_w(c_{vw}+D_{i-1}(w)))$$

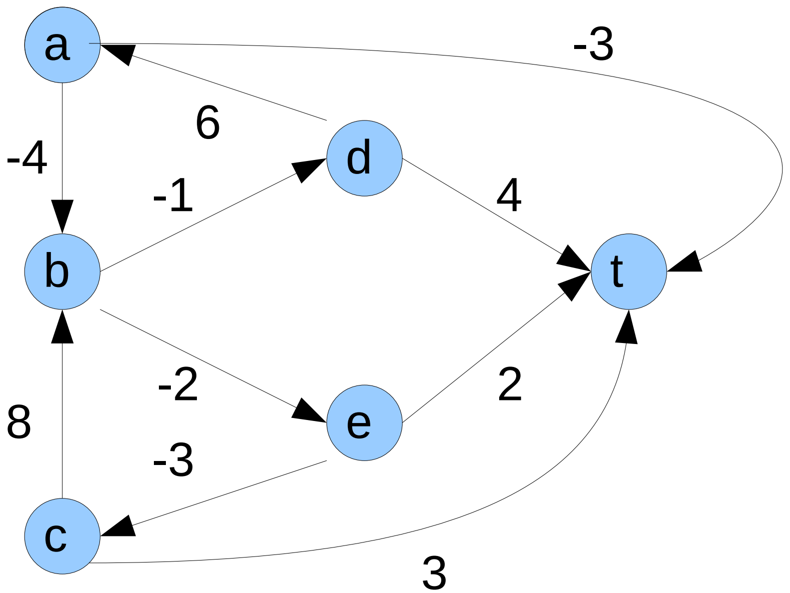
Ejemplo



	0	1	2	3	4	5
t	0	0	0	0	0	
a	inf	-3	-3	-4	-6	
b	inf	inf	0	-2	-2	
c	inf	3	3	3	3	
d	inf	4	3	3	2	
e	inf	2	0	0	0	

$$D_i(v)=\min(D_{i-1}(v), \min_w(c_{vw}+D_{i-1}(w)))$$

Ejemplo



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	inf	-3	-3	-4	-6	-6
b	inf	inf	0	-2	-2	-2
c	inf	3	3	3	3	3
d	inf	4	3	3	2	0
e	inf	2	0	0	0	0

$$D_i(v)=\min(\ D_{i-1}(v),\ \min_w(c_{vw}+D_{i-1}(w)))$$

Algoritmo de Bellman-Ford

```
n= numero de nodos en el grafo
D[t]=0;
Para cada v!=t  D[v]=10e30;
Para i=1 to n-1
   Para cada v
       D[v]=min_w(D[w]+c[v,w]);
Return D[s];
```

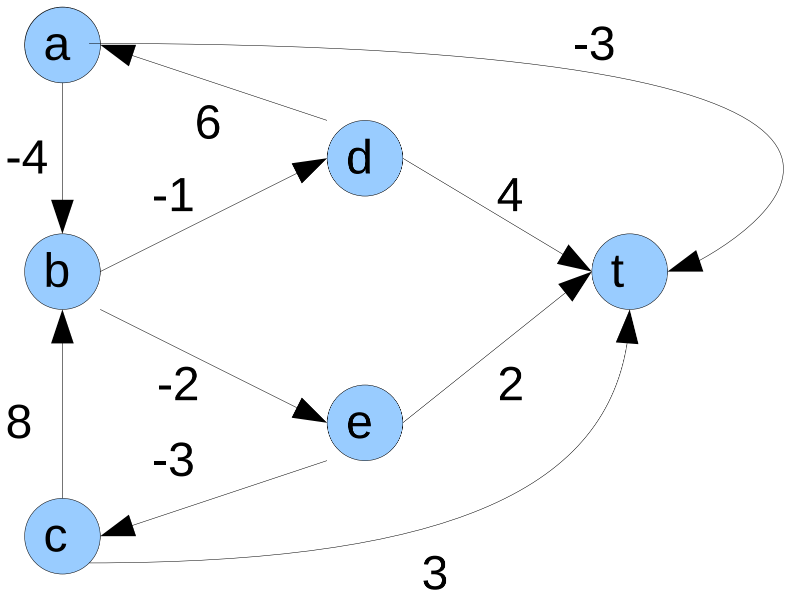
- Realmente calcula el camino mínimo entre cualquier nodo y t.
- No necesita almacenar la matriz, basta con un vector
- Eficiencia: en principio $O(n^3)$ pero se puede hacer que sea $O(nm)$, siendo m el número de arcos, calculando el mínimo solo para aquellos nodos w tales que exista el arco v->w

Para encontrar el camino

```
n= numero de nodos en el grafo
D[t]=0;
Para cada v!=t  D[v]=10e30;
Para i=1 to n-1
   Para cada v
      D[v]=min_w(D[w]+c[v,w]);
      first[v]=w donde se alcanza el minimo
Return D[s];
```

- El camino mínimo de s a t es s->first[s]->first[first[s]]->first[first[first[s]]]->...->t

Ejemplo



	0	1	2	3	4	5	first
t	0	0	0	0	0	0	
a	inf	-3	-3	-4	-6	-6	b
b	inf	inf	0	-2	-2	-2	e
c	inf	3	3	3	3	3	t
d	inf	4	3	3	2	0	a
e	inf	2	0	0	0	0	c

Caminos minimos: a->b->e->c->t,
b->e->c->t,
c->t,
d->a->b->e->c->t,
e->c->t,