Programmazione Dinamica

A.A. 2017-2018

Cammini minimi in un grafo orientato
Moltiplicazione di una sequenza di matrici

Cammini minimi -Cormen

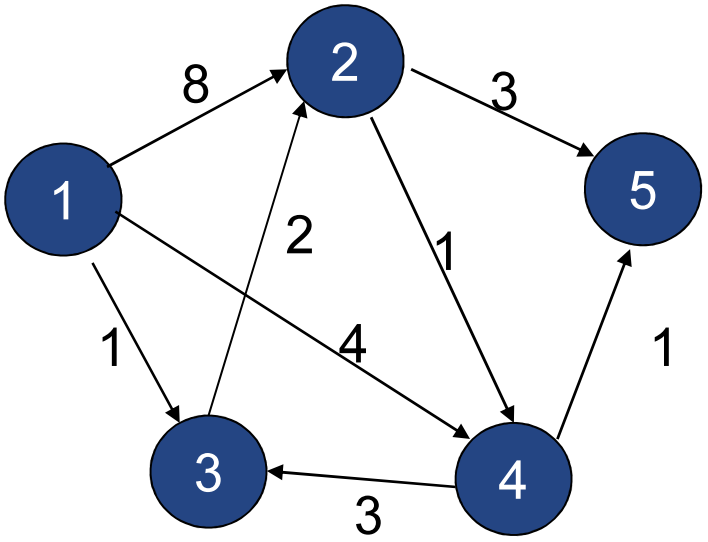
Problema
Trovare il peso dei cammini di peso minimo tra tutte le coppie di vertici di un grafo orientato e pesato

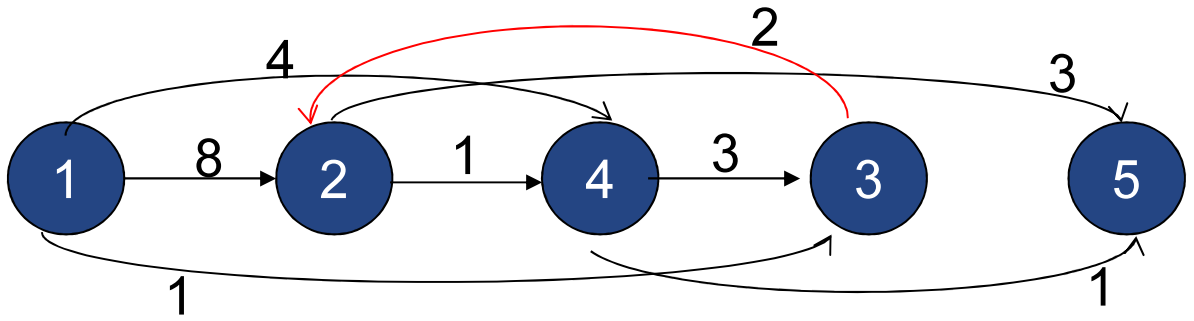
Ipotesi: i grafi non contengono cicli di peso negativo

Ma possono esserci *archi* di peso negativo

Cammini minimi

Nell'esempio che segue non esiste un modo per ordinare i vertici (e i sottoproblemi) in un DAG. Possiamo ancora pensare ad un algoritmo di programmazione dinamica?





Cammini minimi

Si deve pensare al problema in modo alternativo!

Dobbiamo identificare sottoproblemi diversi, in particolare, dato che il grafo è definito da una coppia <V, E>, è naturale pensare ad una definizione induttiva basata su:

- Cammini minimi definiti in base al numero di archi che formano il cammino: i sottoproblemi sono i cammini con un numero di archi minore;

- Cammini minimi definiti in funzione dei vertici che li compongono: i sottoproblemi sono i cammini che attraversano meno vertici.

Cammini minimi: algoritmo di ‘’moltiplicazione”

Cerchiamo una definizione di cammini minimi in funzione della lunghezza dei cammini, basandoci sulle seguenti osservazioni:

- Possiamo esprimere la non esistenza di un cammino con un peso infinito.
- non esiste nessun cammino lungo 0 archi da un vertice i a un vertice j $\neq$ i.
- solo per un *singolo nodo* “i” esiste un cammino lungo 0 da i a se stesso
- il cammino di peso minore tra i e j $\neq$ i di lunghezza al più m è un cammino di lunghezza minore di m, oppure un cammino di lunghezza m, ottenuto da un cammino minimo di lunghezza m -1 da i a un vertice t, seguito dall’arco <t, j>

Cammini minimi: algoritmo di ‘’moltiplicazione”

$$\text{Dist}^{(0)}(i, j) = \begin{cases} 0 & \text{se } j \equiv i \\ \infty & \text{se } j \neq i \end{cases}$$

$$\text{Dist}^{(m)}(i, j) = \min \{\text{Dist}^{(m-1)}(i, j), \min_{k \neq j} \{\text{Dist}^{(m-1)}(i, k) + w(k, j)\}\}$$

Estendendo la funzione peso con la definizione:

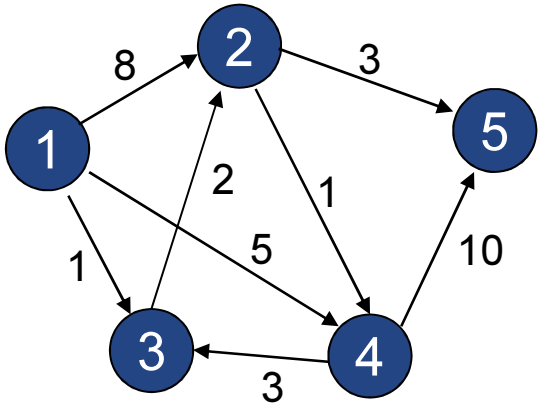
$$w(i,i) = 0 \qquad \forall\, i \in V[G]$$

si può semplificare:

$$\text{Dist}^{(m)}(i, j) = \min_{k \in V[G]} \{\text{Dist}^{(m-1)}(i, k) + w(k, j)\}$$

Cammini minimi: algoritmo di ‘’moltiplicazione’

	1	2	3	4	5
1	0	8	1	5	∞
2	∞	0	∞	1	3
3	∞	2	0	∞	∞
4	∞	∞	3	0	10
5	∞	∞	∞	∞	0



$Dist_1^{(2)}$ | 0 | 3 | 1 | 5 | 11 |

$Dist_1^{(3)}$ | 0 | 3 | 1 | 4 | 6 |

Cammini minimi: algoritmo di ‘’moltiplicazione’

Per trovare i cammini minimi tra tutte le coppie di vertici possiamo usare una matrice $D^{(m)}$ per memorizzare i pesi dei cammini di lunghezza al massimo m.
W sia la matrice che memorizza il grafo.

$$D^{(0)}[i,j] = \begin{cases} 0 & \text{se } i = j \\ \infty & \text{se } i \neq j \end{cases}$$

$$D^{(m)}[i,j]) = \min_{1 \leq k \leq n} \{D^{(m-1)}[i,k]) + W[k,j]) \}$$

Nota: $D^{(1)} = W$

Cammini minimi: algoritmo di ‘’moltiplicazione’

‘’Travasiamo’’ la matrice D in una matrice D’

```
Extend-shortest-paths (D, W)
    D’: matrice n x n
    n ← righe [W]
    for i ← 1 to n
        for j ← 1 to n
            D’[i,j] ← ∞
            for k ← 1 to n
                D’[i,j] ← min (D’[i,j], D[i,k] + W[k,j])
return D’
```

La complessità dell’algoritmo *Extend-shortest-paths* e` $\Theta(n^3)$

Cammini minimi: algoritmo di ‘’moltiplicazione’

Slow-all-pairs-shortest-paths (D, W)

n ← righe [W]

D ← W {W e` la matrice $D^{(1)}$}

for m ← 2 to n-1

D’ ← *Extend-shortest-paths* (D, W)

D ← D’ {D’ e` la matrice $D^{(m)}$}

return D {D e` la matrice $D^{(n-1)}$}

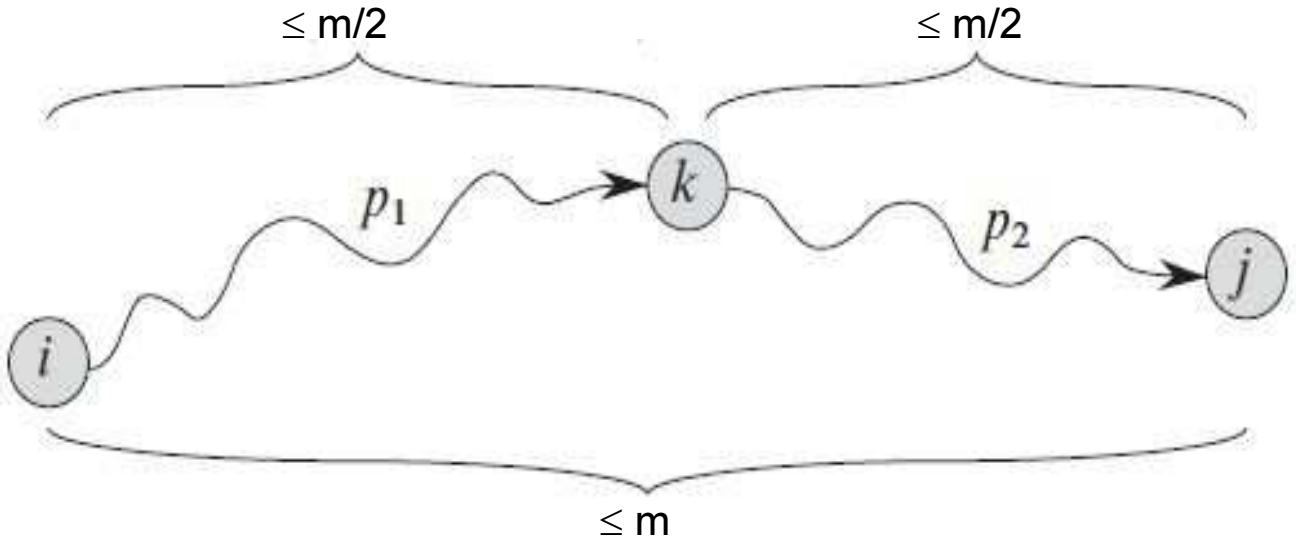
La complessità dell’algoritmo *Slow-all-pairs-shortest-paths e`* $\Theta(n^4)$

Cammini minimi: versione più efficiente

Osservazioni:

- il cammino minimo tra due nodi è lungo al più n-1. Quindi $D^{(k)} = D^{(n-1)}$ per ogni $k \geq n-1$

- Supposto m *pari* un cammino minimo da **i** a **j** di lunghezza al più m sarà scomponibile in un cammino minimo di lunghezza al più m/2 da i a un qualche nodo **k** e un cammino minimo di lunghezza al più m/2 da **k** a **j**.



Cammini minimi: versione più efficiente

Si arriva quindi a questa equazione di ricorrenza (molto simile a quella precedente). In questo caso è più conveniente partire da m=1, cioè $D^{(1)} = W$

$$D^{(1)}[i,j] = W[i,j]$$

$$D^{(m)}[i,j]) = \min_{1 \le k \le n} \{D^{(m/2)}[i,k] + D^{(m/2)}[k,j] \} \quad \text{se } m>1$$

- Anche in questo caso è essenziale impostare la soluzione in forma iterativa.
- Si può usare la procedura *Extend-shortest-paths*(-, -) vista in precedenza, cambiando solo i parametri.

Cammini minimi: versione più efficiente

Faster-all-pairs-shortest-paths (D, W)

n ← righe [W]

D ← W

m ← 1

<u>while</u> (m < n-1)

D' ← *Extend-shortest-paths* (D, D)

m ← 2m

D ← D' {D e' la matrice $D^{(m)}$, D' la matrice $D^{(2m)}$}

<u>return</u> D {D e` la matrice $D^{(h)}$, con h minima potenza di $2 \geq (n-1)$ e $D^{(h)} = D^{(n-1)}$, cioè $D = D^{(n-1)}$}

La complessità dell'algoritmo *Faster-all-pairs-shortest-paths e`* $\Theta(n^3 \log_2 n)$

Cammini minimi: algoritmo di Floyd-Warshall

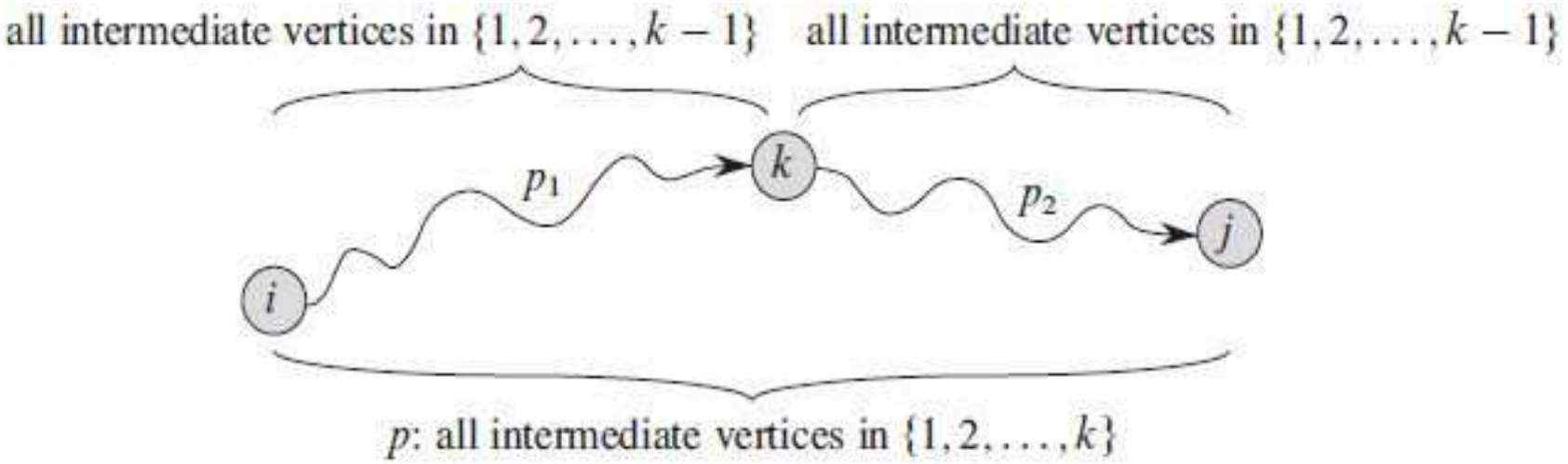
Numeriamo i vertici da 1 a n e definiamo il cammino minimo tra due vertici in funzione dei vertici attraversati dal cammino:

- esiste un cammino tra i e j che attraversa 0 vertici se e solo se esiste l'arco < i,j >

- il cammino di peso minore tra i e j che attraversa al più i vertici 1, 2, …k è un cammino che attraversa al più i vertici 1, 2, …k-1, oppure un cammino che attraversa il vertice k, ottenuto dalla concatenazione di un cammino minimo da i a k e di un cammino minimo da k a j, entrambi con i vertici intermedi nell'insieme {1, 2,…, k-1}.

Cammini minimi: algoritmo di Floyd-Warshall

Ora la riduzione a sottoproblemi si può rappresentare in questo modo:



Cammini minimi: algoritmo di Floyd-Warshall

$$Ds^{(0)}(i, j) = W(i,j)$$

$$Ds^{(m)}(i, j) = \min \{Ds^{(m-1)}(i,j), Ds^{(m-1)}(i,m) + Ds^{(m-1)}(m,j)\}$$

Se anche in questo caso usiamo delle matrici $D^{(m)}$ i loro elementi risultano così definiti:

$$D^{(m)}[i,j] = \begin{cases} W[i,j] & \text{se } m = 0 \\ \min (D^{(m-1)}[i,j], D^{(m-1)}[i,m] + D^{(m-1)}[m,j]) & \text{se } m > 0 \end{cases}$$

Cammini minimi: algoritmo di Floyd-Warshall

```
Floyd-Warshall (W)
    n ← righe [W]
    D ← W                          {W e` la matrice D(0)}
    for m ← 1 to n
        for i ← 1 to n
            for j ← 1 to n
                D'[i,j] ← min {D[i,j], D[i,m] + D[m,j]}
        D ← D'                     {D e` la matrice D(m-1), D' la matrice D(m)}
    return D                       {D e` la matrice D(n)}
```

La complessità dell'algoritmo *Floyd-Warshall* e` $\Theta(n^3)$

Moltiplicazione di matrici

Date n matrici $A_1A_2\ldots A_n$, con A_i di dimensione $c_{i-1} \times c_i$, determinare una parentesizzazione del prodotto $A_1A_2\ldots A_n$ che minimizzi il numero complessivo di moltiplicazioni scalari.

Esempio: *A*: 10 x 100 *B*: 100 x 5 *C*: 5 x 50

Se il calcolo viene effettuato secondo la parentesizzazione: ((*A B*) *C*) si eseguono 7.500 moltiplicazioni scalari (10 x 100 x 5 = 5.000 per la prima moltiplicazione e 10 x 5 x 50 = 2.500 per la seconda), mentre, se viene effettuato secondo la parentesizzazione: (*A* (*B C*)) se ne eseguono 75.000 (100 x 5 x 50 = 25.000 + 10 x 100 x 50 = 50.000).

Il noto algoritmo per moltiplicare due matrici

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2      error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4      for i = 1 to A.rows
5          for j = 1 to B.columns
6              c_ij = 0
7              for k = 1 to A.columns
8                  c_ij = c_ij + a_ik · b_kj
9      return C
```

Moltiplicazione di matrici

Altro esempio: *A*: 100 x 1 *B*: 1 x 100 *C*: 100 x 1

• *((A B) C)* *(A B)* *(A B) C)*	*Numero moltiplicazioni* ***100 x 1 x 100 = 10.000*** ***100 x 100 x1 = 10.000*** ***20.000***
• *(A (B C))* *(B C)* *(A (BC))*	***1 x 100 x 1 = 100*** ***100 x 1 x 1 = 100*** ***200***

Moltiplicazione di matrici

Data una sequenza di n matrici, quante sono le possibili parentesizzazioni?

Per $n = 3$ sono 2, per $n = 4$ sono 5, per $n = 5$?

$P(n)$: numero di parentetizzazioni per n matrici $A_1A_2A_3\ldots A_n$

- L'ultima moltiplicazione può occorrere in $n-1$ posizioni diverse.
- Fissato l'indice k dell'ultima moltiplicazione, abbiamo $P(k)$ parentetizzazioni per $A_1A_2\ldots A_k$ e $P(n-k)$ parentesizzazioni per $A_{k+1}\ldots A_n$

Moltiplicazione di matrici

Otteniamo in tal modo:

$$P(n) = \sum_{k=1}^{n-1} P(k) \cdot P(n-k)$$ (numeri Catalani)

n	1	2	3	4	5	6	7	8	9	10
P(n)	1	1	2	5	14	42	132	429	1430	4862

$P(n) = \Omega(2^n)$ $(P(n) \sqrt{\pi}\, 4^n\, n^{3/2}$

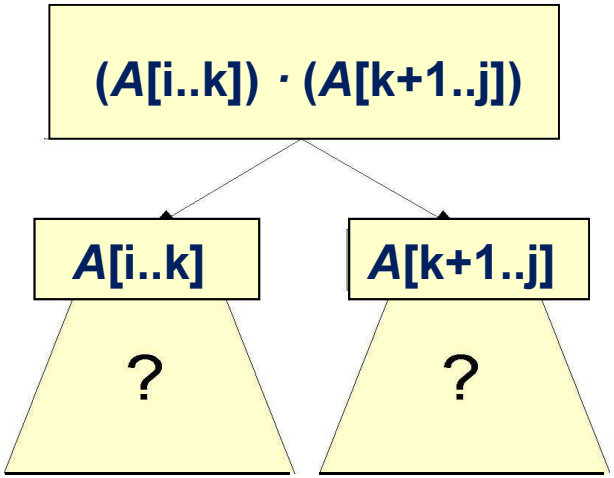
Conseguenza: la “forza bruta” (tentare tutte le possibili parentesizzazioni) è troppo costosa.

Moltiplicazione di matrici

Indichiamo con $A[i..j] = A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$ il prodotto delle matrici della sottosequenza dalla i-esima alla j-esima.
Si consideri una parentesizzazione *ottima* di $A[i..j]$. Esiste una "ultima moltiplicazione": in altre parole, esiste un indice k tale che

$$A[i..j] = A[i..k] \cdot A[k+1..j]$$

Quali sono le caratteristiche delle due sotto-parti $A[i..k]$ e $A[k+1..j]$?
Il problema ha la proprietà delle sottostruttura ottima ?



Sottostruttura ottima.
Se $A[i...j] = A[i...k] \cdot A[k+1...j]$ è una parentesizzazione ottima del prodotto da A_i ad A_j, allora $A[i...k]$ e $A[k+1...j]$ sono parentesizzazioni ottime dei prodotti da A_i ad A_k e da A_{k+1} ad A_j rispettivamente.

Dimostrazione
Ragioniamo per assurdo.
Supponiamo che esista una parentesizzazione ottima $A'[i..k]$ delle matrici $A_i \ldots A_j$ con costo inferiore a $A[i..k]$. Allora, $A'[i...k] \cdot A[k+1...j]$ sarebbe una parentesizzazione di $A_i \ldots A_j$ con costo inferiore a $A[i..j]$, assurdo.

Moltiplicazione di matrici

Definiamo induttivamente il costo di una soluzione ottima.

Sia m(i,j) il *numero minimo di prodotti scalari* richiesti per calcolare il prodotto $A_i A_{i+1} \ldots A_j$. Come calcolare m(i,j)?

Caso base: i = j, allora m(i,j) = 0
Passo induttivo: i < j.
La parentesizzazione ottima: A[i...j]=A[i...k] ·A[k+1...j], suggerisce la relazione:

$$m(i,j) = m(i,k) + m(k+1,j) + c_{i-1} \cdot c_k \cdot c_j$$

- $m(i,k)$: no minimo prodotti per moltiplicare $A_i, \ldots, A_k$
- $m(k+1,j)$: Num. minimo prodotti per moltiplicare $A_{k+1}, \ldots, A_j$
- $c_{i-1} \cdot c_k \cdot c_j$: Numero prodotti per moltiplicare le 2 matrici

Ma non conosciamo il valore di k ... possiamo provarli tutti!
Poiché *k* può assumere i valori fra *i* e *j-1*, si ottiene la formula finale:

$$m(i,j) = \min_{i \le k < j} \{m(i,k) + m(k+1,j) + c_{i-1} \cdot c_k \cdot c_j\}$$

Moltiplicazione di matrici

La definizione ricorsiva di *m*(i,j) suggerisce di utilizzare un approccio ricorsivo top-down per risolvere il problema. **Proviamo...**

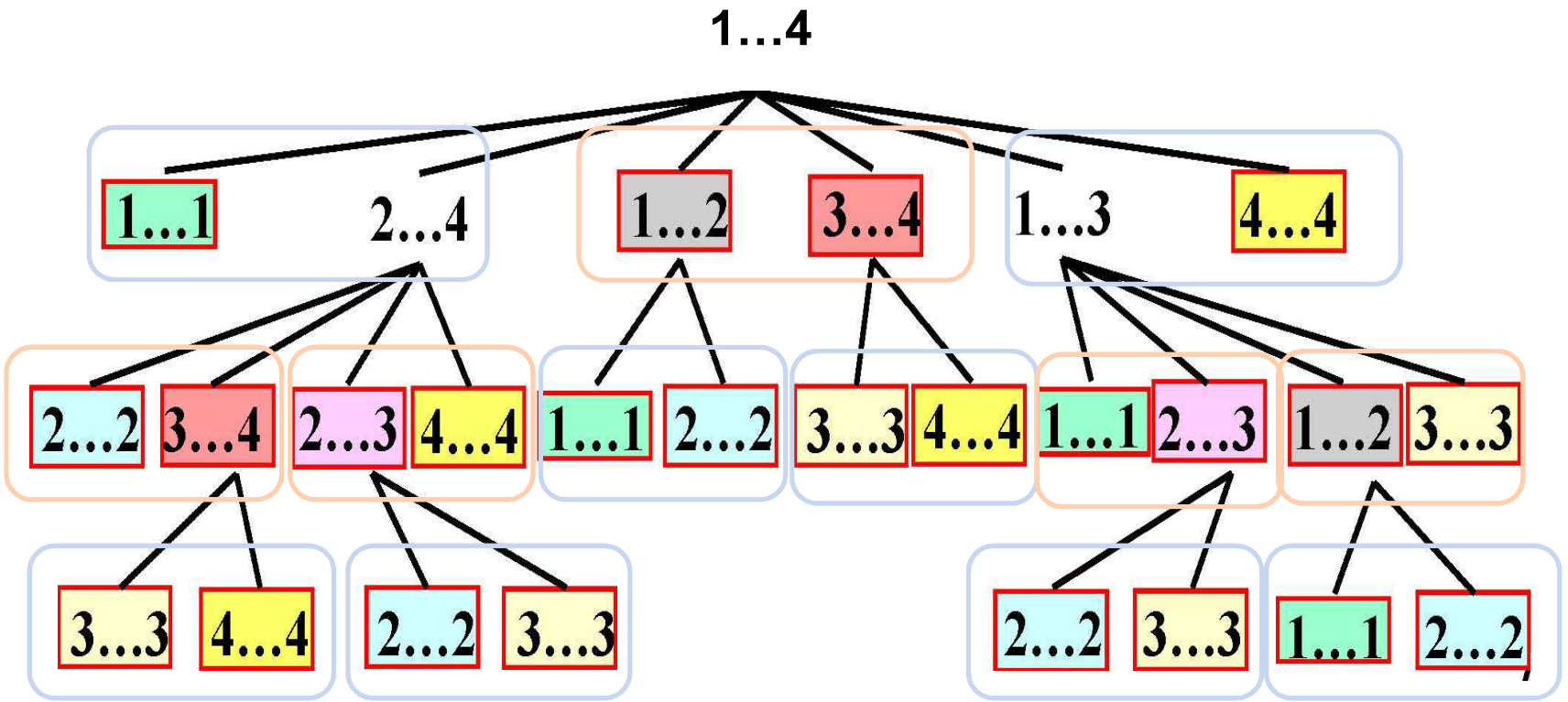
Input: un array c[0..n] con le dimensioni delle matrici;

```
recursive_matrix_chain (c, i, j)
    if (i = j)  return 0
    min ← ∞
    for k ← i to j -1
            q ←  recursive_matrix_chain(c, i, k) +
                 + recursive_matrix_chain(c, k+1, j) +
                 + c[i-1] * c[k] * c[j]
            if (q < min)   min ← q
    return min
```

Complessità risultante?

Moltiplicazione di matrici

- La soluzione ricorsiva top-down è $\Omega(2^n)$
- Non è migliore dell'approccio basato su forza bruta!
- Il problema principale è che molti problemi vengono risolti più volte



Moltiplicazione di matrici

Ma il numero di problemi è molto inferiore a 2^n, uno per ogni scelta di i e j con $1 \le i \le j \le n$:

$$\binom{n}{2} + n = \Theta(n^2)$$

Sottoproblemi con $i \neq j$

Sottoproblemi con $i = j$

Mai calcolare più di una volta la soluzione ad un sottoproblema!

Programmazione dinamica

Moltiplicazione di matrici

“calcolare in modo bottom-up il valore della soluzione ottima”

Abbiamo bisogno di una matrice m: n x n per ricordare i valori dei sottoproblemi che vengono risolti.
Ricordiamo comunque che il prodotto viene effettuato sempre tra una matrice di indice inferiore e una di indice superiore, pertanto ci serve solo il triangolo superiore della matrice.
La diagonale principale, relativa al numero di moltiplicazioni in presenza di una sola matrice, viene inizializzata a 0.

Come sono legati i sottoproblemi?

Moltiplicazione di matrici

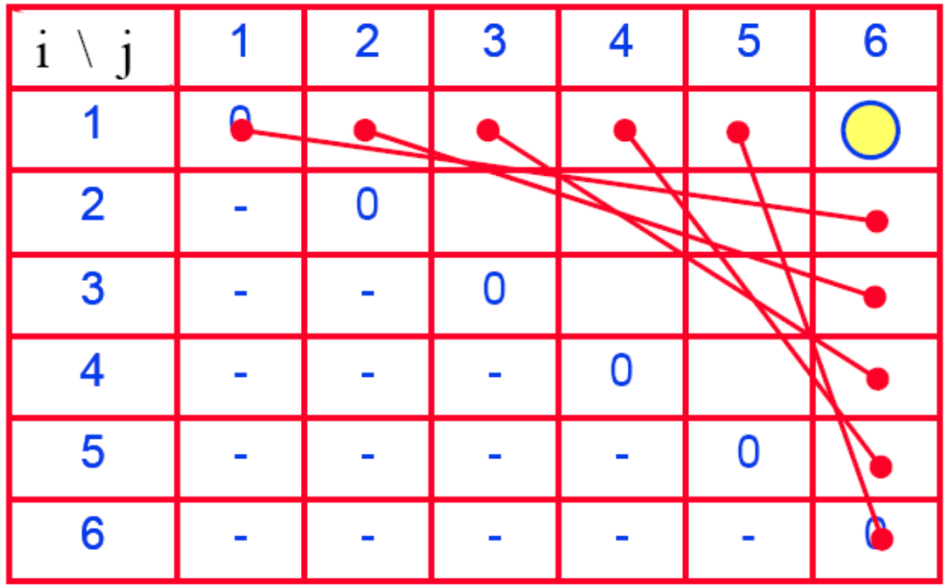
i \ j	1	2	3	4	5	6
1	0					
2	-	0				
3	-	-	0			
4	-	-	-	0		
5	-	-	-	-	0	
6	-	-	-	-	-	0

i \ j	1	2	3	4	5	6
1	0					
2	-	0				
3	-	-	0			
4	-	-	-	0		
5	-	-	-	-	0	
6	-	-	-	-	-	0

$m[1,2] = \min_{1 \le k < 2}\{m[1,k] + m[k+1,2] + c_0.c_k.c_2\} =$
$= m[1,1] + m[2,2] + c_0 \cdot c_k \cdot c_2 =$
$= c_0 \cdot c_k \cdot c_2$

$m[2,5] = \min_{2 \le k < 5}\{m[2,k] + m[k+1,5] + c_1.c_k.c_5\} =$
$= \min \{m[2,2] + m[3,5] + c_1 \cdot c_2 \cdot c_5 ,$
$m[2,3] + m[4,5] + c_1 \cdot c_3 \cdot c_5 ,$
$m[2,4] + m[5,5] + c_1 \cdot c_4 \cdot c_5\}$

Moltiplicazione di matrici



i \ j	1	2	3	4	5	6
1	0					
2	-	0				
3	-	-	0			
4	-	-	-	0		
5	-	-	-	-	0	
6	-	-	-	-	-	0

$$m[1,6] = \min_{1 \le k < 6}\{m[1,k] + m[k+1,6] + c_0 \cdot c_k \cdot c_6\} =$$

$$= \min \{m[1,1] + m[2,6] + c_0 \cdot c_1 \cdot c_6 \, , \, m[1,2] + m[3,6] + c_0 \cdot c_2 \cdot c_6 \, ,$$

$$m[1,3] + m[4,6] + c_0 \cdot c_3 \cdot c_6 \, , \, m[1,4] + m[5,6] + c_0 \cdot c_4 \cdot c_6 \, ,$$

$$m[1,5] + m[6,6] + c_0 \cdot c_5 \cdot c_6\}$$

Moltiplicazione di matrici

```
Matrix-chain-order (c)
    n ← length (c) -1
    for i ← 1 to  n
        m[i, i] ← 0
    for d ← 2 to  n
        for tutti gli elementi [i, j] con
        j-i = d-1 (diagonale d-1)
            m[i, j] ← ∞
            for k ← i to  j-1
                q ← m[i, k] + m[k+1, j] + c[i-1].c[k].c[j]
                if (q < m[i, j])   m[i, j] ← q
    return m[1,n]
```

d è la lunghezza della sequenza

Moltiplicazione di matrici

```
Matrix-chain-order (c)
    n ← length (c) -1
    for i ← 1 to  n
        m[i, i] ← 0        s[i, i] ← 0
    for d ← 2 to  n
        for i ← 1 to  n-d+1
            j ← i+d-1
            m[i, j] ← ∞
            for k ← i to  j-1
                q ← m[i, k] + m[k+1, j] + c[i-1].c[k].c[j]
                if (q < m[i, j])  m[i, j] ← q
    return m[1,n ]
```

d è la lunghezza della sequenza

i e j assumono i valori delle coordinate degli elementi sulla diagonale d (j-i=d-1)

La matrice s ricorda il valore di k che rende minimo il numero di moltiplicazioni scalari

La complessità dell'algoritmo *Matrix-chain-order* è $O(n^3)$

Applichiamo l'algoritmo a una sequenza di 6 matrici le cui dimensioni sono fornite dal vettore c: 7 8 4 2 3 5 6

	1	2	3	4	5	6	
1	0	224	176	218	276	350	1
2	-	0	64	112	174	250	2
3	-	-	0	24	70	138	3
4	-	-	-	0	30	90	4
5	-	-	-	-	0	90	5
6	-	-	-	-	-	0	6

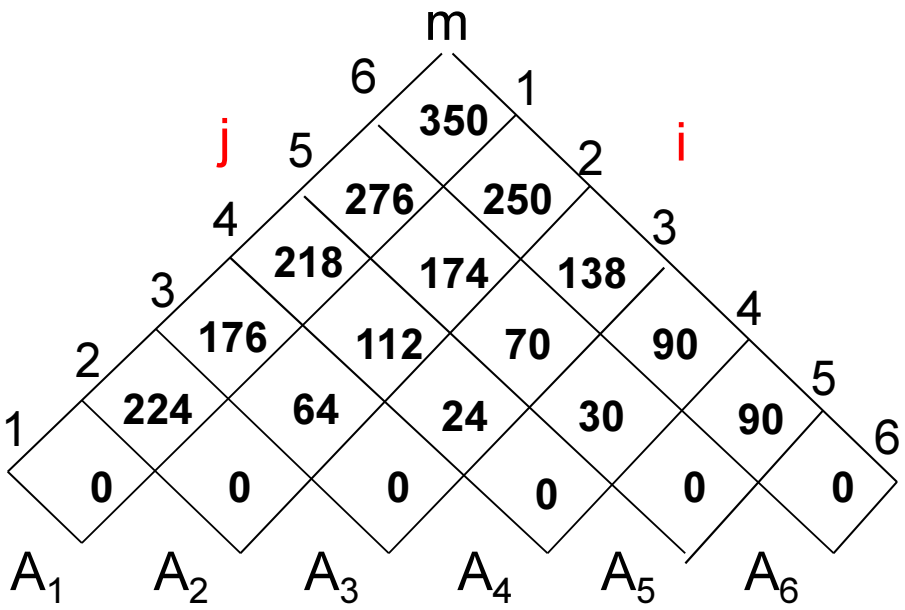
Matrice m

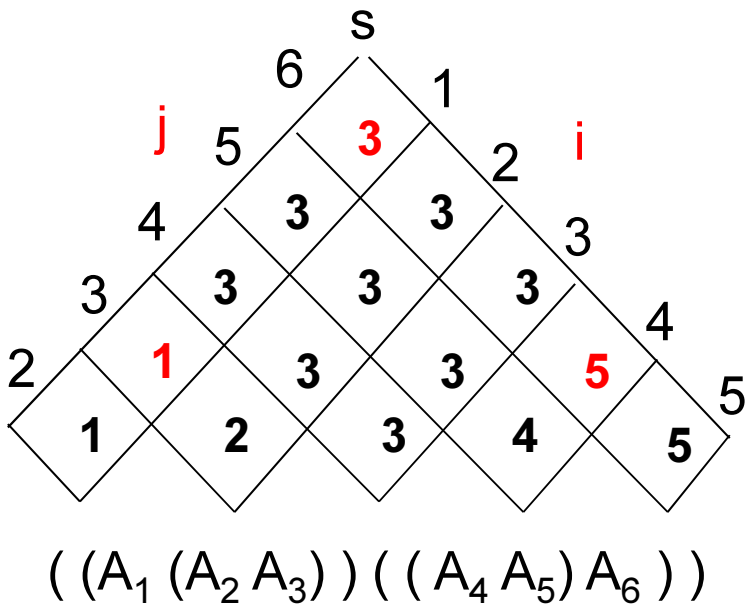
	1	2	3	4	5	6
1	0	1	1	3	3	3
2	-	0	2	3	3	3
3	-	-	0	3	3	3
4	-	-	-	0	4	5
5	-	-	-	-	0	5
6	-	-	-	-	-	0

Matrice s

Moltiplicazione di matrici

Con una rotazione di 45° a sinistra si ottiene:





((A1 (A2 A3)) ((A4 A5) A6))

Moltiplicazione di matrici

La sequenza di moltiplicazioni è facilmente realizzabile con un algoritmo ricorsivo:

```
{i <= j}
Matrix-chain-multiply (A, s, i, j)
    if (j > i)  k ←  s[i, j]
                X ← Matrix-chain-multiply (A, s, i, k)
                Y ← Matrix-chain-multiply (A, s, k+1, j)
                return Matrix-multiply (X, Y)
    else return A[i]
```

Moltiplicazione di matrici

```
Matrix-chain-print (s, i, j)
    if (j > i)  k ← s[i, j]
                print "("
                Matrix-chain-print (s, i, k)
                print "·"
                Matrix-chain-print (s, k+1, j)
                print ")"
    else print "A_i"
```

Per l'esempio precedente (c : 7 8 4 2 3 5 6) si ottiene:

$$(((A_1 \cdot (A_2 \cdot A_3)) \cdot ((A_4 \cdot A_5) \cdot A_6)))$$