Tecniche Algoritmiche/2
Programmazione Dinamica

Moreno Marzolla
Dip. di Scienze dell'Informazione
Università di Bologna

marzolla@cs.unibo.it
http://www.moreno.marzolla.name/

Original work Copyright © Alberto Montresor, Università di Trento, Italy
(http://www.dit.unitn.it/~montreso/asd/index.shtml)
Modifications Copyright © 2010—2011 Moreno Marzolla, Università di Bologna, Italy
(http://www.moreno.marzolla.name/teaching/ASD2010/)

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/2.5/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Programmazione dinamica

Richard Ernest Bellman (1920—1984), http://en.wikipedia.org/wiki/Richard_Bellman

- Definizione
  - Strategia di programmazione sviluppata negli anni '50 da Richard E. Bellman
  - Ambito: problemi di ottimizzazione
  - Trovare la soluzione ottima secondo un "indice di qualità" assegnato ad ognuna delle soluzioni possibili
- Approccio
  - Risolvere un problema combinando le soluzioni di sotto-problemi
  - Ma ci sono importanti differenze con divide-et-impera

Programmazione dinamica vs divide-et-impera

- Divide-et-impera
  - Tecnica ricorsiva
  - Approccio top-down (problemi divisi in sottoproblemi)
  - Vantaggioso solo quando i sottoproblemi sono indipendenti
    - Altrimenti gli stessi sottoproblemi possono venire risolti più volte
- Programmazione dinamica
  - Tecnica iterativa
  - Approccio bottom-up
  - Vantaggiosa quando ci sono sottoproblemi in comune
- Esempio semplice: i numeri di Fibonacci

Esempio:

Numero di Fibonacci
(ancora loro!)

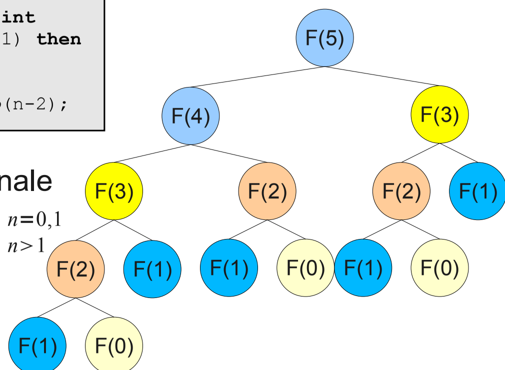
Numeri di Fibonacci

http://en.wikipedia.org/wiki/Fibonacci_number

- Definiti ricorsivamente
  - *F(0) = F(1) = 1*
  - *F(n) = F(n-2)+F(n-1)*
- In natura:
  - Pigne, conchiglie, parte centrale dei girasoli, etc.
- In informatica:
  - Alberi AVL minimi, Heap di Fibonacci, etc.

Implementazione ricorsiva

```
algoritmo Fib(int n) → int
  if (n == 0) or (n == 1) then
    return 1;
  else
    return Fib(n-1)+Fib(n-2);
```



- Costo computazionale

$$T(n)=\begin{cases} O(1) & n=0,1 \\ T(n-1)+T(n-2)+O(1) & n>1 \end{cases}$$

- Soluzione
  - $T(n) = O(2^n)$

Implementazione iterativa

```
algoritmo Fib(int n) → int
  f := new array[0..n] of int
  f[0] := 1 ; f[1] := 1
  for i := 2 to n do
    f[i] := f[i-1] + f[i-2];
  endfor
  return f[n];
```

- Complessità
  - Tempo: O(n)
  - Spazio: O(n)

n	0	1	2	3	4	5	6	7
f []	1	1	2	3	5	8	13	21

Implementazione iterativa - 2

```
algoritmo Fib(int n) → int
  if (n<2) then
    return 1;
  else
    f := new array[0..2] of int;
    f[1] := 1; f[2] := 1;
    for i := 2 to n do
      f[0] = f[1];
      f[1] = f[2];
      f[2] := f[1] + f[0];
    endfor
    return f[2];
  endif
```

- Complessità
  - Tempo: $O(n)$
  - Spazio: $O(1)$
    - Array di 3 elementi

n	2	3	4	5
f[0]	1	1	2	3
f[1]	1	2	3	5
f[2]	2	3	5	8

Quando applicare la programmazione dinamica?

- Sottostruttura ottimale
  - È possibile combinare le soluzioni dei sottoproblemi per trovare la soluzione di un problema più grande
    - PS: In tempo polinomiale!
  - Le decisioni prese per risolvere un problema rimangono valide quando esso diviene un sottoproblema di un problema più grande
- Sottoproblemi ripetuti
  - Un sottoproblema può occorrere più volte
- Spazio dei sottoproblemi
  - Deve essere polinomiale

Programmazione dinamica

1) Caratterizzare la struttura di una soluzione ottima
2) Definire ricorsivamente il valore di una soluzione ottima
   - La soluzione ottima ad un problema contiene le soluzioni ottime ai sottoproblemi
3) Calcolare il valore di una soluzione ottima “bottom-up” (cioè calcolando prima le soluzioni ai casi più semplici)
   - Si usa una tabella per memorizzare le soluzioni dei sottoproblemi
   - Evitare di ripetere il lavoro più volte, utilizzando la tabella
4) Costruire la (una) soluzione ottima.

Esempio:

Sottovettore di valore massimo

Esempio
sottovettore di valore massimo

- Consideriamo un vettore V[] di n elementi (positivi o negativi che siano)
- Vogliamo individuare il sottovettore di V la cui somma di elementi sia massima

3	-5	10	2	-3	1	4	-8	7	-6	-1

- Domanda: quanti sono i sottovettori di V?
  - 1 sottovettore di lunghezza n
  - 2 sottovettori di lunghezza n-1
  - 3 sottovettori di lunghezza n-2
  - ...
  - k sottovettori di lunghezza n-k+1
  - ...
  - n sottovettori di lunghezza 1

Totale: n(n-1)/2 =$\Theta(n^2)$

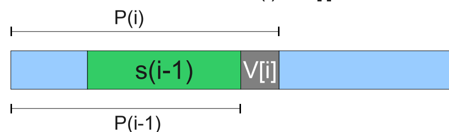
Approccio / 1

- Sia P(i) il problema che consiste nel determinare il valore massimo della somma degli elementi dei sottovettori del vettore V[1..i] che hanno V[i] come ultimo elemento
- Sia s(i) il valore della soluzione di P(i)
  - quindi s(i) è il valore massimo della somma dei sottovettori di V[1..i] che hanno V[i] come ultimo elemento
- La soluzione s* al problema di partenza può essere espressa come

$$s^* = max_{1 \le i \le n} s(i)$$

Approccio / 2

- P(1) ammette una unica soluzione
  - s(1) = V[1]
- Consideriamo P(i)
  - Supponiamo di avere già risolto P(i-1), e quindi di conoscere s(i-1)
  - Se s(i-1)+V[i] ≥ V[i]
    - allora la soluzione ottima è s(i) = s(i-1)+V[i]
  - Se s(i-1)+V[i] < V[i]
    - allora la soluzione ottima è s(i) = V[i]



Esempio

V[]	3	-5	10	2	-3	1	4	-8	7	-6	-1
S[]	3	-2	10	12	9	10	14	6	13	7	6

S[i] = max { V[i], V[i]+S[i-1] }

L'algoritmo

```
algoritmo sottovettoreMax(V, n) → intero
   S := new array [1..n] di interi;
   S[1] := V[1];
   max := 1;
   for i:=2 to n do
      if ( S[i-1]+V[i]≥V[i] ) then
         S[i] := S[i-1]+V[i];
      else
         S[i] := V[i];
      endif
      if ( S[i]>S[max] ) then
         max := i;
      endif
   endfor
   return S[max];
```


Come individuare il sottovettore?

- Fino ad ora siamo in grado di calcolare il valore della *massima somma* di un sottovettore di V[]
- Come facciamo a determinare *quale* sottovettore produce tale somma?
  - L'indice finale del sottovettore già ce l'abbiamo
  - L'indice iniziale lo possiamo ricavare procedendo "a ritroso"
    - Se S[i] = V[i], il sottovettore massimo inizia nella posizione i

Esempio

V[]	3	-5	10	2	-3	1	4	-8	7	-6	-1
S[]	3	-2	10	12	9	10	14	6	13	7	6

```
algoritmo indiceInizio(V, S, max) → intero
   i := max;
   while( S[i] != V[i] ) do
      i := i-1;
   endwhile
   return i;
```


Esempio:

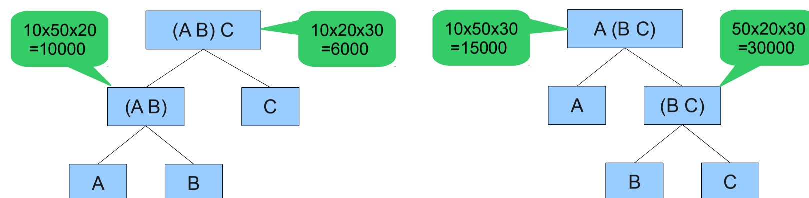
Catena di moltiplicazione di matrici

Catena di moltiplicazione di matrici

- Problema:
  - Data una sequenza di matrici A_1, A_2, A_3, …, A_n, compatibili 2 a 2 al prodotto, vogliamo calcolare il loro prodotto.
- Cosa vogliamo ottimizzare
  - La moltiplicazione di matrici si basa sulla moltiplicazione scalare come operazione elementare.
  - Vogliamo calcolare il prodotto impiegando il numero minore possibile di moltiplicazioni
- Attenzione:
  - Il prodotto di matrici non è commutativo...
  - ...ma è associativo: $(A_1 \times A_2) \times A_3 = A_1 \times (A_2 \times A_3)$

Catena di moltiplicazione tra matrici

- A=matrice n*m, B=matrice m*p
  - AB è una matrice n*p, per calcolare la quale occorrono n*m*p moltiplicazioni
- Esempio
  - A matrice 10x50, B matrice 50x20, C matrice 20x30



Tot: 16000 moltiplicazioni

Tot: 45000 moltiplicazioni

Catena di moltiplicazione tra matrici

- 4 matrici:

A	B	C	D
50x10	10x40	40x30	30x5

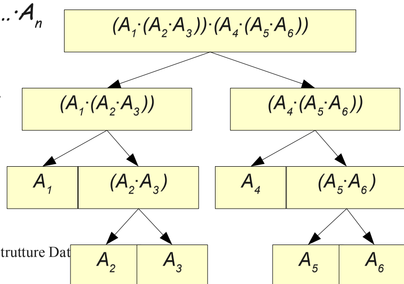
- (((A B) C) D) : 87500 moltiplicazioni
- ((A (B C)) D) : 34500 moltiplicazioni
- ((A B)(C D)) : 36000 moltiplicazioni
- (A ((B C) D)) : 16000 moltiplicazioni
- (A (B (C D))) : 10500 moltiplicazioni

Applicare la programmazione dinamica

- Le fasi principali:
  1. Caratterizzare la struttura di una soluzione ottima
  2. Definire ricorsivamente il valore di una soluzione ottima
  3. Calcolare il valore di una soluzione ottima “bottom-up” (dal basso verso l’alto)
  4. Costruzione di una soluzione ottima
- Nei lucidi successivi:
  - Vediamo ora una ad una le quattro fasi del processo di sviluppo applicate al problema della parentesizzazione ottima

Parentesizzazione

- Definizione: Un prodotto di matrici $A_1 \cdot A_2 \cdot A_3 \ldots \cdot A_n$ si dice *completamente parentesizzato* se:
  - consiste di un'unica matrice ($n = 1$), oppure
  - per qualche $1 \le k < n$, è il prodotto, delimitato da parentesi, tra i prodotti completamente parentesizzati $A_1 \cdot A_2 \cdot A_3 \cdot \ldots \cdot A_k$ e $A_{k+1} \cdot A_{k+2} \cdot A_{k+3} \cdot \ldots \cdot A_n$
- Esempio:
  - $(A_1 \cdot (A_2 \cdot A_3)) \cdot (A_4 \cdot (A_5 \cdot A_6)) \rightarrow k=3$
  - "Ultima moltiplicazione"



Parentesizzazione ottima

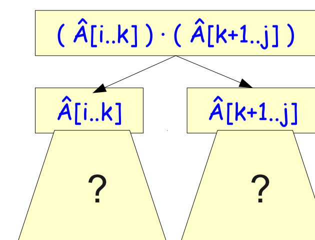
- Parentesizzazione ottima
  - Determinare il numero di moltiplicazioni scalari necessari per i prodotti tra le matrici in ogni parentesizzazione
  - Scegliere una delle parentesizzazioni che richiedono il numero minimo di moltiplicazioni complessivamente
- Motivazione:
  - Vale la pena di spendere un po' di tempo per cercare la parentesizzazione migliore, per risparmiare tempo dopo
- Si puo' dimostrare che il numero P(n) di possibili parentesizzazioni di n matrici è $\Omega(2^n)$
  - ...quindi la "forza bruta" non è praticabile

Definizioni

- Denoteremo nel seguito con:
  - $A_1 \cdot A_2 \cdot A_3 \cdot \ldots \cdot A_n$ il prodotto di n matrici da ottimizzare
  - c_{i-1} il numero di righe della matrice A_i
  - c_i il numero di colonne della matrice A_i
  - $A[i..j]$ il prodotto $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$
  - $\hat{A}[i..j]$ una parentesizzazione di $A[i..j]$ (non necessariamente ottima)

Struttura di una parentesizzazione ottima

- Sia $A[i..j] = A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$ una sottosequenza del prodotto di matrici
  - Si consideri una parentesizzazione *ottima* $\hat{A}[i..j]$ di $A[i..j]$
  - Esiste una "ultima moltiplicazione": in altre parole, esiste un indice k tale che $\hat{A}[i..j] = \hat{A}[i..k] \cdot \hat{A}[k+1..j]$
- Domanda:
  - Quali sono le caratteristiche delle due sotto-parti $\hat{A}[i..k]$ e $\hat{A}[k+1..j]$?



Struttura di una parentesizzazione ottima

- Teorema (sottostruttura ottima)
  - Se Â[i..j] = Â[i..k] · Â[k+1..j] è una parentesizzazione ottima del prodotto A[i..j], allora Â[i..k] e Â[k+1..j] sono parentesizzazioni ottime dei prodotti A[i..k] e A[k+1..j], rispettivamente.
- Dimostrazione
  - Ragionamento per assurdo
  - Supponiamo esista un parentesizzazione ottima Â'[i..k] di A[i..k] con costo inferiore a Â[i..k]
  - Allora, Â'[i..k] · Â[k+1..j] sarebbe una parentesizzazione di A[i..j] con costo inferiore a Â[i..j], assurdo.

Struttura di una parentesizzazione ottima

- In altre parole:
  - Il teorema afferma che esiste una sottostruttura ottima: *Ogni soluzione ottima al problema della parentesizzazione contiene al suo interno le soluzioni ottime dei due sottoproblemi*
- Programmazione dinamica:
  - L'esistenza di sottostrutture ottime è una delle caratteristiche da cercare per decidere se la programmazione dinamica è applicabile
- Prossima fase:
  - Definire ricorsivamente il costo di una soluzione ricorsiva

Definire ricorsivamente il valore di una soluzione ottima

- Definizione: sia *m[i,j]* il numero minimo di prodotti scalari richiesti per calcolare il prodotto $A[i,j] = A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$
- Come calcolare *m[i,j]*?
  - Caso base: $i=j$. Allora, $m[i,j] = 0$
  - Passo ricorsivo: $i < j$. Esiste una parentesizzazione ottima *Â[i..j] = Â[i..k] · Â[k+1..j]*; sfruttiamo la ricorsione:

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

Prodotti per Â[i..k]

Prodotti per Â[k+1..j]

c_{i-1} : # righe prima matrice

c_k : # colonne prima matrice

c_j : # colonne seconda matrice

Definire ricorsivamente il valore di una soluzione ottima

- Ma qual è il valore di *k*?
  - Non lo conosciamo....
  - ... ma possiamo provarli tutti!
  - *k* può assumere valori fra *i* e *j-1*
- La formula finale:
  - $m[i,j] = \min_{i \le k < j} \{ m[i,k] + m[k+1,j] + c_{i-1} \cdot c_k \cdot c_j \}$

Esempio

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

$m[1,2] = min_{1 \leq k<2} \; \{ m[1,k] + m[k+1,2] + c_1 c_k c_2 \}$
$= m[1,1] + m[2,2] + c_0 c_1 c_2$
$= c_0 c_1 c_2$

Esempio

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

$m[2,4] = min_{2 \leq k<4} \{ m[2,k] + m[k+1,4] + c_1 c_k c_4 \}$
$= min \{ \; m[2,2] + m[3,4] + c_1 c_2 c_4 ,$
$m[2,3] + m[4,4] + c_1 c_3 c_4 \; \}$

Esempio

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

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

Esempio

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

$m[1,5] = min_{1 \leq k<5} \{ m[1,k] + m[k+1,5] + c_0 c_k c_5 \}$
$= min \{ \; m[1,1] + m[2,5] + c_0 c_1 c_5 ,$
$m[1,2] + m[3,5] + c_0 c_2 c_5 ,$
$m[1,3] + m[4,5] + c_0 c_3 c_5 ,$
$m[1,4] + m[5,5] + c_0 c_4 c_5 \; \}$

Esempio

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

$m[1,6] = \min_{1 \le k < 6} \{ m[1,k] + m[k+1,6] + c_0 c_k c_6 \}$
$= \min \{ m[1,1] + m[2,6] + c_0 c_1 c_6 ,$
$m[1,2] + m[3,6] + c_0 c_2 c_6 ,$
$m[1,3] + m[4,6] + c_0 c_3 c_6 ,$
$m[1,4] + m[5,6] + c_0 c_4 c_6 ,$
$m[1,5] + m[6,6] + c_0 c_5 c_6 \}$

Calcolo “bottom-up” del valore della soluzione

- Terzo passo della programmazione dinamica:
  - *“calcolare in modo bottom-up il valore della soluzione ottima”*
- La definizione ricorsiva di *m[i,j]* suggerisce di utilizzare un approccio ricorsivo top-down:
  - Lanciamo il problema sulla sequenza completa *[1,n]*
  - Il meccanismo ricorsivo individua i sottoproblemi da risolvere
- Proviamo...
  - Input: un array *c[0..n]* con le dimensioni delle matrici,
    - *c[0]* è il numero di righe della prima matrice
    - *c[i]* è il numero di colonne della matrice A_i

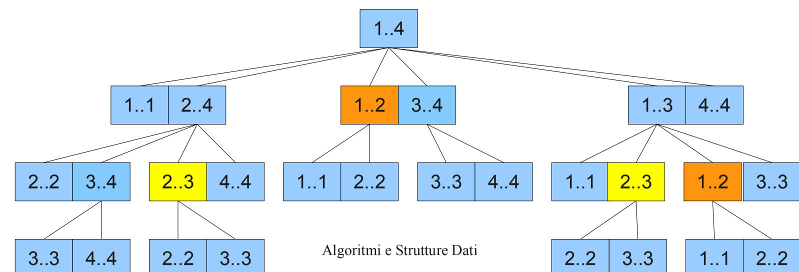
Soluzione ricorsiva top-down

```
algoritmo recursive-matrix-chain(array c[0..n], i, j)
  if (i==j) then
    return 0;
  endif
  min := +∞ ;
  for k := i to j-1 do
    q = recursive-matrix-chain(c, i, k)+
        recursive-matrix-chain(c, k+1, j)+
        c[i-1]*c[k]*c[j];
    if (q < min) then
      min := q;
    endif
  endfor
  return min;
```

Non efficiente!

Critica all'approccio top-down

- Alcune riflessioni
  - La soluzione ricorsiva top-down è $\Omega(2^n)$ (si può dimostrare per induzione che $T(n) \ge c2^n$)
  - Non è poi migliore dell'approccio basato su forza bruta!
- Molti sottoproblemi vengono risolti più volte



Calcolare la soluzione ottima in modo bottom-up

- È interessante notare che il numero di possibili 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≠j

Sottoproblemi con i=j

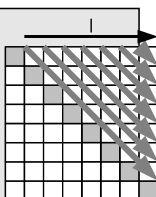
- Ogni sottoproblema
  - È risolvibile utilizzando le soluzioni dei sottoproblemi che sono state eventualmente già calcolate e memorizzate nell'array
- Idea chiave della programmazione dinamica:
  - Mai calcolare più di una volta la soluzione ad un sottoproblema

Calcolare la soluzione ottima in modo bottom-up

- L'algoritmo MCO (Matrix-Chain-Order)
  - prende in ingresso un array *c[0..n]* con le dimensioni delle matrici
    - *c[0]* è il numero di righe della prima matrice
    - *c[i]* è il numero di colonne della matrice A_i
  - utilizza (e ritorna) due matrici $n \cdot n$ ausiliarie:
    - *m[i,j]* che contiene i costi minimi dei sottoproblemi *A[i..j]*
    - *s[i,j]* che contiene il valore di *k* che minimizza il costo per il sottoproblema *A[i..j]*

Soluzione ottima in modo bottom-up

```
algoritmo MCO(c[0..n])
  for i := 1 to n do
     m[i,i] := 0;
     s[i,i] := 0;
  endfor
  for l := 2 to n do
    for i := 1 to n - l + 1 do
      j := i + l - 1;
      m[i,j] := ∞ ;
      for k := i to j - 1 do
        q := m[i,k] + m[k+1,j] + c[i-1]c[k]c[j];
        if (q < m[i,j]) then
          m[i,j] := q;
          s[i,j] := k;
        endif
      endfor
    endfor
  endfor
  return m[], s[]
```



m[ ]

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

i	c_i
0	7
1	8
2	4
3	2
4	3
5	5
6	6

$m[1,4] = \min_{1 \le k \le 3} \{ m[1,k] + m[k+1,4] + c_0 c_k c_4 \}$

$= \min\{ m[1,1] + m[2,4] + c_0 c_1 c_4,\ m[1,2] + m[3,4] + c_0 c_2 c_4,\ m[1,3] + m[4,4] + c_0 c_3 c_4 \}$

$= \min\{ 0 + 112 + 7 * 8 * 3,\ 224 + 24 + 7 * 4 * 3,\ 176 + 0 + 7 * 2 * 3 \}$

$= \min\{ 280, 332, 218 \} = 218$

s[ ]

i \ j	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

i	c_i
0	7
1	8
2	4
3	2
4	3
5	5
6	6

$m[1,4[= \min_{1 \le k \le 3} \{ m[1,k] + m[k+1,4] + c_0 c_k c_4 \}$

$= \min\{ m[1,1] + m[2,4] + c_0 c_1 c_4,$

$m[1,2] + m[3,4] + c_0 c_2 c_4,$

$m[1,3] + m[4,4] + c_0 c_3 c_4 \}$

$= \min\{ 0 + 112 + 7 * 8 * 3,$

$224 + 24 + 7 * 4 * 3,$

$176 + 0 + 7 * 2 * 3 \}$

$= \min\{ 280, 332, 218 \} = 218$

Calcolare la soluzione ottima in modo bottom-up

- Considerazioni sull'algoritmo
  - Costo computazionale: $O(n^3)$
- Nota
  - Lo scopo della terza fase era “calcolare in modo bottom-up il valore della soluzione ottima”
  - Questo valore si trova in *m[1,n]*
- Per alcuni problemi
  - È anche necessario mostrare la soluzione trovata
  - Per questo motivo registriamo informazioni sulla soluzione mentre procediamo in maniera bottom-up

Costruire una soluzione ottima

- Possiamo definire un algoritmo che costruisce la soluzione a partire dall'informazione calcolata da MCO.
  - La matrice *s[ ]* ci permette di determinare il modo migliore di moltiplicare le matrici.
  - *s[i,j]* contiene infatti il valore di *k* su cui dobbiamo spezzare il prodotto *A[i..j]*
  - Ci dice cioè che per calcolare *A[i..j]* dobbiamo prima calcolare *A[i..k]* e *A[k+1..j]* e poi moltiplicarle tra loro.
- Ma questo è un processo facilmente realizzabile tramite un algoritmo ricorsivo

Costruire una soluzione ottima

- Algoritmo 1: prodotto

```
algoritmo MCM(A[], s[], i, j) → matrice
  if (j > i) then
    k := s[i,j];
    X := MCM(A, s, i, k);
    Y := MCM(A, s, k + 1, j);
    return matrix-multiplication(X,Y);
  else
    return A[i];
  endif
```

Costruire una soluzione ottima

- Algoritmo 2: stampa

```
algoritmo MCP(s[], i, j)
  if (j > i) then
    k = s[i,j];
    print "(";
    MCP(s, i, k);
    print "·";
    MCP(s, k+1, j);
    print ")";
  else
    print "A[",i,"]";
  endif
```


Esempio di esecuzione

$A_{1..6} = A_{1..k} \times A_{k+1..6}$
$= A_{1..3} \times A_{4..6}$

$A_{1..3} = A_{1..k} \times A_{k+1..3}$
$= A_1 \times A_{2..3}$

$A_{4..6} = A_{4..k} \times A_{k+1..6}$
$= A_{4..5} \times A_6$

$A_{2..3} = A_{2..k} \times A_{k+1..3}$
$= A_2 \times A_3$

$A_{4..5} = A_{4..k} \times A_{k+1..5}$
$= A_4 \times A_5$

s[ ]

i \ j	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

$A_{1..6} = ((A_1 \times (A_2 \times A_3)) \times ((A_4 \times A_5) \times A_6))$