Programmazione dinamica vs. divide-et-impera

Analogia

Sono entrambi paradigmi di sintesi di algoritmi che risolvono problemi combinando le soluzioni di sottoproblemi

Differenza

Secondo divide-et-impera si suddivide in sottoproblemi che vengono risolti uno indipendentemente dall'altro (si dice che i sottoproblemi sono tutti indipendenti) → eventuali sottoproblemi comuni vengono risolti più volte

Secondo la programmazione dinamica ciascun sottoproblema viene risolto una sola volta, la sua soluzione viene memorizzata in una tabella e riusata se si ripresenta lo stesso sottoproblema (→ si dice che la programmazione dinamica è un metodo di soluzione tabulare e che si applica anche a sottoproblemi non indipendenti, dove due problemi non sono indipendenti se tutti e due richiedono la soluzione di almeno un sottoproblema comune)

Programmazione dinamica

Viene generalmente adottata per risolvere problemi di ottimizzazione, ciascuno dei quali ammette solitamente più soluzioni ottime

Un algoritmo che adotta questo paradigma si articola in quattro fasi, di cui l'ultima è opzionale:

1) Caratterizzazione della struttura di una soluzione ottima
2) Definizione ricorsiva del valore di una soluzione ottima
3) Calcolo del valore di una soluzione ottima con una strategia bottom-up
4) Costruzione di una soluzione ottima a partire dalle informazioni calcolate al punto precedente

Il paradigma viene illustrato di seguito mediante un esempio: l'ottimizzazione del prodotto di una sequenza di matrici

Prodotto di una sequenza di matrici

$A = \langle A_1, A_2, .. , A_n \rangle$ = sequenza di matrici

Soluzione del problema del prodotto
Usare l'algoritmo standard del prodotto di due matrici (di cui al prossimo lucido), dopo avere fissato in modo disambiguo, mediante parentesi, l'ordine dei prodotti da effettuare

Un prodotto di matrici è completamente parentesizzato se

- consiste in un'unica matrice, oppure
- è il prodotto, delimitato da parentesi, di due prodotti di matrici completamente parentesizzati

Prodotto di due matrici

```
MATRIX-MULTIPLY(A, B)
1   if columns[A] ≠ rows[B]                          O(1)
2       then   error "dimensioni non compatibili"
3       else   for i ← 1 to rows[A]
4                  do for j ← 1 to columns[B]
5                         do  C[i, j] ← 0
6                             for k ← 1 to columns[A]
7                                 do  C[i, j] ← C[i, j] + A[i, k] · B[k, j]
8   return C
```

$C_{p,r} = A_{p,q} B_{q,r} \rightarrow$
n° totale dei prodotti scalari eseguiti alla linea 7 per il calcolo di C = pqr

Prodotto di più matrici: costo computazionale

Parentesizzazioni distinte portano allo stesso risultato, grazie alla proprietà associativa del prodotto matriciale, ma comportano costi di calcolo diversi

Esempio

matrici	A_1	A_2	A_3
dimensioni	10×100	100×5	5×50

Parentesizzazione	**N° prodotti scalari**
$((A_1\ A_2)\ A_3)$	$10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 5000 + 2500 = 7500$
$(A_1\ (A_2\ A_3))$	$100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 25000 + 50000 = 75000$

Prodotto di una sequenza di matrici: riformulazione come problema di ottimizzazione

Data una sequenza di n matrici $\langle A_1, A_2, .. , A_n \rangle$ dove A_i ha dimensioni $p_{i-1} \times p_i$ con i = 1, 2, .. , n, determinare una parentesizzazione completa del prodotto che minimizzi il n° complessivo di moltiplicazioni scalari

↓

problema affrontato mediante programmazione dinamica:
determinazione della parentesizzazione ottima

Numero di parentesizzazioni

P(n) = n° di parentesizzazioni distinte di una sequenza di n matrici

$$P(n) = \begin{cases} 1 & se\ n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & se\ n \geq 2 \end{cases}$$

↓

soluzione della ricorrenza: P(n) = C(n – 1) = sequenza dei numeri Catalani dove

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega\left(4^n / n^{3/2}\right)$$

↓

determinare la parentesizzazione ottima mediante una ricerca esaustiva (brutale, ingenua) è inefficiente

Parentesizzazione: caratterizzazione della struttura della soluzione ottima

Convenzione notazionale: $A_{i..j} = A_i\ A_{i+1} \ldots\ A_j$

Parentesizzazioni ottime

Parentesizzazione ottima di $A_1\ A_2 \ldots\ A_n = (A_1\ A_2 \ldots\ A_k)\ (A_{k+1}\ A_{k+2} \ldots\ A_n) = A_{1..k}\ A_{k+1..n} = A_{1..n}$ per qualche intero $k \mid 1 \leq k < n$

Esistenza di sottostrutture ottime all'interno di una soluzione ottima = primo segno distintivo e criterio di applicabilità della programmazione dinamica

Parentesizzazione: definizione ricorsiva della soluzione ottima

Convenzione notazionale: m[i, j] = n° minimo di moltiplicazioni scalari necessarie per calcolare $A_{i..j}$ = costo della soluzione ottima di un sottoproblema

$m[i, i] = 0$ per $i = 1, 2, .. , n$

Costo del calcolo del prodotto $A_{i..k}\ A_{k+1..j}$

$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$ per $i < j,\ k = i, i + 1, .. , j - 1$

ricorrenza:

$$m[i, j] = \begin{cases} 0 & se\ i = j \\ \min_{i \le k < j} \{ m[i,k] + m[k+1, j] + p_{i-1} p_k p_j \} & se\ i < j \end{cases}$$

Convenzione notazionale: s[i, j] = k che produce la parentesizzazione ottima di $A_{i..j}$

Parentesizzazione: calcolo ricorsivo del costo della soluzione ottima

n° totale di sottoproblemi distinti (del problema della parentesizzazione ottima) = un (sotto)problema per ogni possibile scelta di i e j, $1 \leq i \leq j \leq n$, per un totale di

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

n° sottoproblemi di dimensione ≥ 2 (i < j)

n° sottoproblemi di dimensione unitaria (i = j)

Un algoritmo ricorsivo per il calcolo del costo della soluzione ottima (del problema della parentesizzazione) basato sulla ricorrenza m[i, j] del lucido precedente è esponenziale nel tempo (come quello di ricerca esaustiva)
Tale algoritmo, nei diversi cammini del suo albero di ricorsione, può richiedere di risolvere più di una volta lo stesso sottoproblema = secondo segno distintivo e criterio di applicabilità della programmazione dinamica (a cui ricorriamo di seguito)

Parentesizzazione: calcolo bottom-up del costo della soluzione ottima

Ingresso

$p = \langle p_0, p_1, .. , p_n \rangle$ = vettore contenente le dimensioni delle n matrici, dove length[p] = n + 1

Uscite

m[1..n, 1..n] = tabella ausiliaria per la memorizzazione dei costi m[i, j]

s[1..n, 1..n] = tabella ausiliaria per la memorizzazione degli indici k per cui m[i, j] è un costo ottimo

Parentesizzazione: calcolo bottom-up del costo della soluzione ottima (cont.)

```
MATRIX-CHAIN-ORDER(p)
1   n ← length[p] – 1
2   for i ← 1 to n
3       do m[i, i] ← 0
4   for l ← 2 to n
5   ▷ l è la lunghezza della sequenza di matrici del sottoproblema
    considerato
6       do for i ← 1 to n – l + 1
7          do  j ← i + l – 1
8              m[i, j] ← ∞
9              for k ← i to j – 1
10                 do  q ← m[i, k] + m[k + 1, j] + p[i – 1]p[k]p[j]
11                 if q < m[i, j]
12                    then     m[i, j] ← q
13                             s[i, j] ← k
14  return m e s
```

$O(n^3)$

ma si può dimostrare che è anche $\Omega(n^3)$ ($\rightarrow$ è $\Theta(n^3)$)

La quantità di memoria richiesta da m e s è $\Theta(n^2)$

Esempio: MATRIX-CHAIN-ORDER

Matrice	Dimensione
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

$\rightarrow p = \langle 30, 35, 15, 5, 10, 20, 25 \rangle$

Esempio: MATRIX-CHAIN-ORDER (cont.)

$l = 2$

m

i \ j	1	2	3	4	5	6
1	0	15750				
2		0	2625			
3			0	750		
4				0	1000	
5					0	5000
6						0

s

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

Esempio: MATRIX-CHAIN-ORDER (cont.)

$l = 3$

m

i \ j	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625	4375		
3			0	750	2500	
4				0	1000	3500
5					0	5000
6						0

s

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

Esempio: MATRIX-CHAIN-ORDER (cont.)

$l = 4$

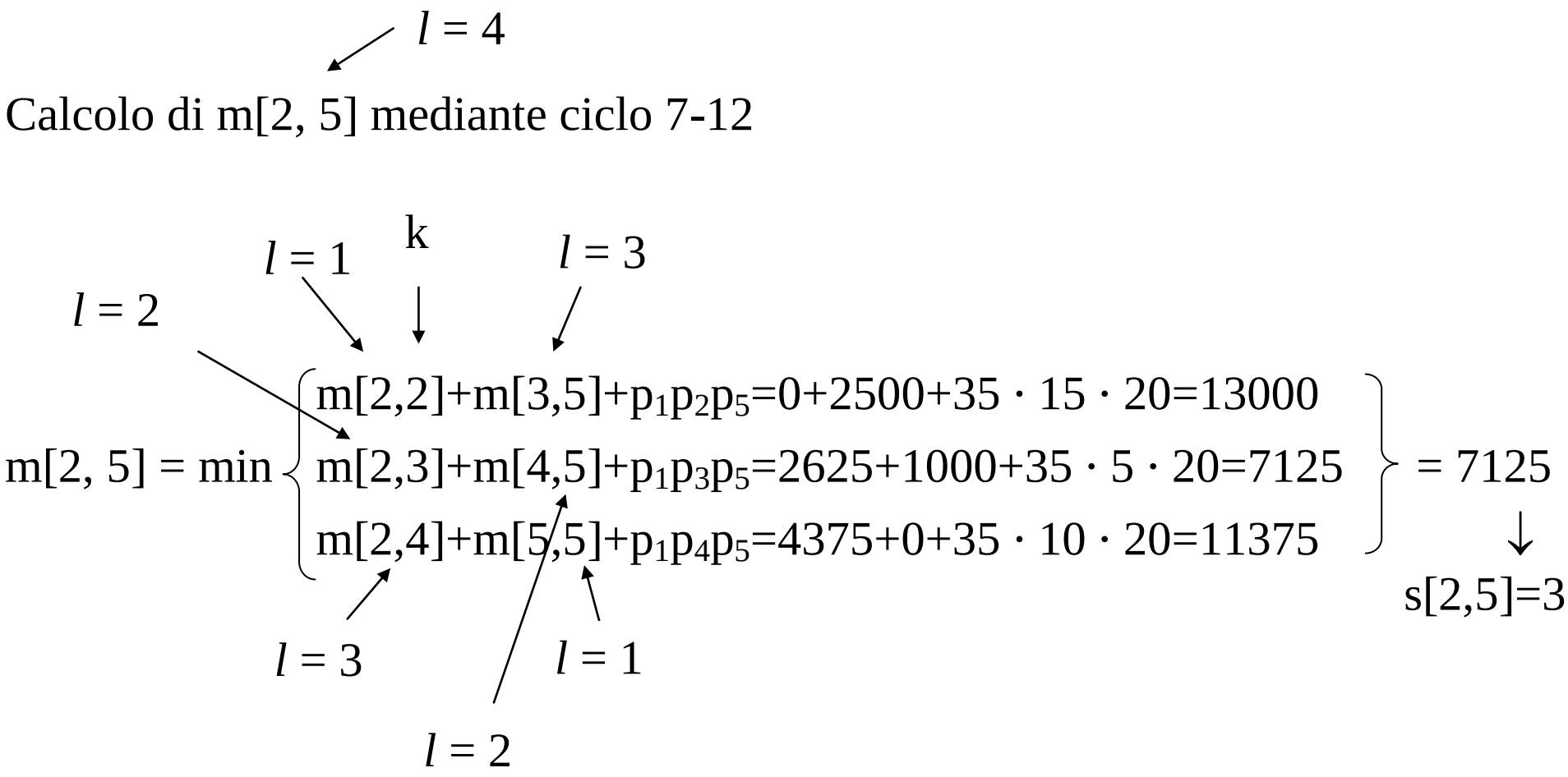
m

i \ j	1	2	3	4	5	6
1	0	15750	7875	9375		
2		0	2625	4375	7125	
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

s

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

Esempio: MATRIX-CHAIN-ORDER (cont.)



Esempio: MATRIX-CHAIN-ORDER (cont.)

$l = 5$

m

i \ j	1	2	3	4	5	6
1	0	15750	7875	9375	11875	
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

s

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

Esempio: MATRIX-CHAIN-ORDER (cont.)

$l = 6$

m

i \ j	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

n° minimo di moltiplicazioni scalari necessarie per effettuare il prodotto delle 6 matrici

s

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

parentesizzazione ottima:
$((A_1(A_2A_3))((A_4A_5)A_6)$

Costruzione di una soluzione ottima

```
MATRIX-CHAIN-MULTIPLY(A, s, i, j)
0 ▷ A = ⟨A1, A2, .. , An⟩ è una sequenza di matrici,
    i e j sono gli indici iniziale e finale della sottosequenza
    di matrici di cui il programma calcola il prodotto
    sfruttando la parentesizzazione ottima descritta nella
    matrice s generata da MATRIX-CHAIN-ORDER
1 if j > i
2   then   X ← MATRIX-CHAIN-MULTIPLY(A, s, i, s[i, j])
3          Y ← MATRIX-CHAIN-MULTIPLY(A, s, s[i, j] + 1, j)
4          return MATRIX-MULTIPLY(X, Y)
5   else   return Ai
```