

Algoritmi e Strutture Dati

Programmazione dinamica – Parte 3

Alberto Montresor

Università di Trento

2025/03/22

This work is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License.



Sommario

- 1 String matching approssimato
- 2 Prodotto di catena di matrici
- 3 Insieme indipendente di intervalli pesati

String matching approssimato

Definizione

Siano:

- $P = p_1 \dots p_m$ una stringa detta **pattern**
- $T = t_1 \dots t_n$ una stringa detta **testo**, con $m \leq n$,

Un'**occorrenza k -approssimata** di P in T è una copia di P in T in cui sono ammessi k "errori" tra P e T , del seguente tipo:

- 1 i corrispondenti caratteri in P, T sono diversi (**sostituzione**)
- 2 un carattere in P non è incluso in T (**inserimento**)
- 3 un carattere in T non è incluso in P (**cancellazione**)

Esempio (Sostituzione, cancellazione)

$T =$ questoèun**o**s**c**empio

$P =$ un**e**sempio

Esempio

Problema – Approximated string matching

Trovare un'occorrenza k -approssimata di P in T con k minimo ($0 \leq k \leq m$).

Spiegazione

- Qual è il minimo valore k per cui si trova un'occorrenza k -approssimata di P in T ?
- Dove?
- Con quali errori?

Sottostruttura ottima

Definizione

Sia $DP[0 \dots m][0 \dots n]$ una tabella di programmazione dinamica tale che $DP[i][j]$ sia il minimo valore k per cui esiste un'occorrenza k -approssimata di $P(i)$ in $T(j)$ che termina nella posizione j

Sottostruttura ottima

Definizione

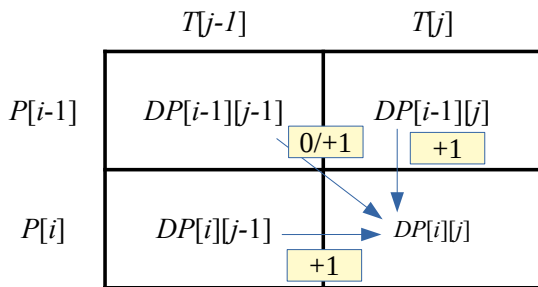
Sia $DP[0 \dots m][0 \dots n]$ una tabella di programmazione dinamica tale che $DP[i][j]$ sia il minimo valore k per cui esiste un'occorrenza k -approssimata di $P(i)$ in $T(j)$ che termina nella posizione j

Quattro possibilità

$DP[i - 1][j - 1]$, se $P[i] = T[j]$	avanza su entrambi i caratteri (uguali)
$DP[i - 1][j - 1] + 1$, se $P[i] \neq T[j]$	avanza su entrambi i caratteri (sost.)
$DP[i - 1][j] + 1$	avanza sul pattern (inserimento)
$DP[i][j - 1] + 1$	avanza sul testo (cancellazione)

Sottostruttura ottima

$$DP[i][j] = \begin{cases} 0 & i = 0 \\ i & j = 0 \\ \min\{DP[i-1][j-1] + \delta, & \delta = \text{üif}(P[i] = T[j], 0, 1) \\ DP[i-1][j] + 1, \\ DP[i][j-1] + 1\} & \text{altrimenti} \end{cases}$$



Dove si trova la soluzione finale?

- $DP[m][j] = k$ se e solo se esiste un'occorrenza k -approssimata di P in $T(j)$ che termina nella posizione j .
- La soluzione del problema è data dal più piccolo valore $DP[m][j]$, per $0 \leq j \leq n$

		T				
		A	B	A	B	A
P		0	0	0	0	0
		1	1	0	1	0
B		2	1	0	1	0
A		3	2	1	1	0
B		2	1	1	0	1

Algoritmo

```

int stringMatching(ITEM[] P, ITEM[] T, int m, int n)


---


int[][] DP = new int[0...m][0...n]
for j = 0 to n do DP[0][j] = 0                                % Caso base: i = 0
for i = 1 to m do DP[i][0] = i                                % Caso base: j = 0
for i = 1 to m do                                           % Caso generale
    for j = 1 to n do
        DP[i][j] = min(DP[i - 1][j - 1] + iif(P[i] == T[j], 0, 1),
                       DP[i - 1][j] + 1,
                       DP[i][j - 1] + 1)
int pos = 0                                                  % Calcola minimo ultima riga
for j = 1 to n do
    if DP[m][j] < DP[m][pos] then
        pos = j
return pos

```

String matching approssimato

Take-home message – prendi e porta a casa

- Non è detto che la "soluzione finale" si trovi nella casella "in basso a destra";
- È invece possibile che la soluzione debba essere ricercata essa stessa nella tabella *DP*

Reality check

Approximate String Matching è un esempio di **string metric**:

[...] is a metric that measures distance ("inverse similarity") between two strings [...]

String metrics are used heavily in information integration and are currently used in areas including fraud detection, fingerprint analysis, plagiarism detection, ontology merging, DNA analysis, RNA analysis, image analysis, evidence-based machine learning, database data deduplication, data mining, incremental search, data integration, and semantic knowledge integration.

https://en.wikipedia.org/wiki/String_metric

Esempi

Edit distance, detta anche **distanza di Levenshtein**

Sommario

- 1 String matching approssimato
- 2 Prodotto di catena di matrici**
- 3 Insieme indipendente di intervalli pesati

Prodotto di catena di matrici

Problema

Data una sequenza di n matrici $A_1, A_2, A_3, \dots, A_n$, compatibili due a due al prodotto, vogliamo calcolare il loro prodotto.

- Il prodotto di matrici non è **commutativo**
- ...ma è **associativo**: $(A_1 \cdot A_2) \cdot A_3 = A_1 \cdot (A_2 \cdot A_3)$

Cosa vogliamo ottimizzare

- Il prodotto di matrici si basa sulla **moltiplicazione scalare** come operazione elementare
- Vogliamo calcolare il prodotto delle n matrici impiegando il più basso numero possibile di moltiplicazioni scalari

Esempio 1

A	100×1
B	1×100
C	100×1

	# Moltiplicazioni	Memoria
$(A \cdot B)$	$100 \times 1 \times 100 = 10000$	10000
$((A \cdot B) \cdot C)$	$\underline{100 \times 100 \times 1 = 10000}$ 20000	$\underline{100}$ 10100
$(B \cdot C)$	$1 \times 100 \times 1 = 100$	1
$(A \cdot (B \cdot C))$	$\underline{100 \times 1 \times 1 = 100}$ 200	$\underline{100}$ 101

Esempio 2

A	50×10
B	10×40
C	40×30
D	30×5

$(((A \cdot B) \cdot C) \cdot D) : 87500$ moltiplicazioni

$((A \cdot (B \cdot C)) \cdot D) : 34500$ moltiplicazioni

$((A \cdot B) \cdot (C \cdot D)) : 36000$ moltiplicazioni

$(A \cdot ((B \cdot C) \cdot D)) : 16000$ moltiplicazioni

$(A \cdot (B \cdot (C \cdot D))) : 10500$ moltiplicazioni

$(((A \cdot B) \cdot C) \cdot D) : 87500$

$(A \cdot B) \quad 50 \times 10 \times 40 = 20000$

$((A \cdot B) \cdot C) \quad 50 \times 40 \times 30 = 60000$

$((A \cdot B) \cdot C) \cdot D \quad \underline{50 \times 30 \times 5 = 7500}$

87500

Parentesizzazione

Parentesizzazione

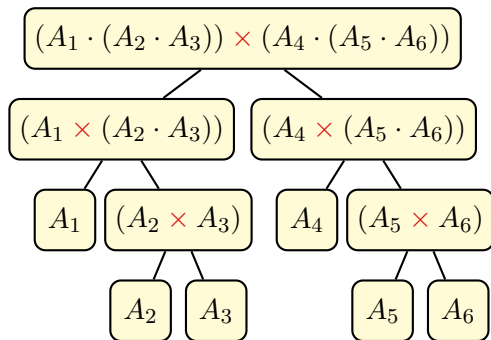
Una **parentesizzazione** $P_{i,j}$ del prodotto $A_i \cdot A_{i+1} \cdots A_j$ consiste:

- nella matrice A_i , se $i = j$;
- nel prodotto di due parentesizzazioni $(P_{i,k} \cdot P_{k+1,j})$, altrimenti.

Esempio

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

In questo caso, $k = 3$ e il prodotto evidenziato è detto "**ultimo prodotto**"



Parentesizzazione ottima

Parentesizzazione ottima

La parentesizzazione che richiede il minor numero di moltiplicazioni scalari per essere completata, fra tutte le parentesizzazioni possibili.

Motivazione

Vale la pena preprocessare i dati per cercare la parentesizzazione migliore, per risparmiare tempo dopo nel calcolo vero e proprio

Domanda

Quante sono le parentesizzazioni possibili?

n	1	2	3	4	5	6	7	8	9	10
$P(n)$	1	1	2	5	?	?	?	?	?	?

Parentesizzazione ottima

- $P(n)$: numero di parentesizzazioni per n matrici $A_1 \cdot \dots \cdot A_n$
- L'ultimo prodotto può occorrere in $n - 1$ posizioni diverse
- Fissato l'indice k dell'ultimo prodotto, abbiamo:
 - $P(k)$ parentesizzazioni per $A_1 \cdot \dots \cdot A_k$
 - $P(n - k)$ parentesizzazioni per $A_{k+1} \cdot \dots \cdot A_n$

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

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

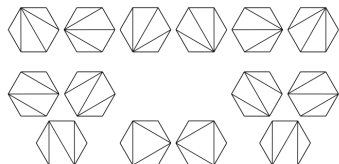
Parentesizzazione ottima

Numero di Catalan

$$P(n) = C(n) = \frac{1}{n+1} \binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} = \Theta\left(\frac{4^n}{n\sqrt{n}}\right)$$

In matematica

$C(n)$: numero di modi in cui un poligono convesso con $n + 2$ lati può essere suddiviso in triangoli.



Esercizio

Dimostrare che $P(n) = \Omega(2^n)$

Implicazione

Algoritmi di forza bruta non vanno quindi bene

Definizioni matematiche

$A_1 \cdot A_2 \cdot \dots \cdot A_n$	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
$A[i \dots j]$	il sottoprodotto $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$
$P[i \dots j]$	una parentesizzazione per $A[i \dots j]$ (non necessariamente ottima)

Struttura di una parentesizzazione ottima

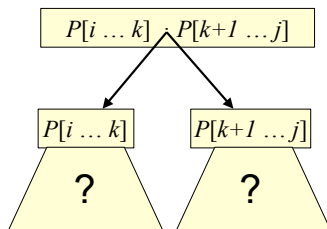
Osservazioni

- Sia $A[i \dots j]$ una sottosequenza del prodotto di matrici
- Si consideri una parentesizzazione ottima $P[i \dots j]$ di $A[i \dots j]$
- Esiste un **ultimo prodotto**: esiste un indice k tale che

$$P[i \dots j] = P[i \dots k] \cdot P[k + 1 \dots j]$$

Domanda

Quali sono le caratteristiche dei due sottoprodotti $P[i \dots k]$ e $P[k + 1 \dots j]$?



Teorema sottostruttura ottima

Teorema

Se $P[i \dots j] = P[i \dots k] \cdot P[k+1 \dots j]$ è una parentesizzazione ottima del prodotto $A[i \dots j]$, allora:

- $P[i \dots k]$ è parentesizzazione ottima del prodotto $A[i \dots k]$
- $P[k+1 \dots j]$ è parentesizzazione ottima del prodotto $A[k+1 \dots j]$

Dimostrazione – per assurdo

- Supponiamo esista un parentesizzazione ottima $P'[i \dots k]$ di $A[i \dots k]$ con costo inferiore a $P[i \dots k]$.
- Allora, $P'[i \dots k] \cdot P[k+1 \dots j]$ sarebbe una parentesizzazione di $A[i \dots j]$ con costo inferiore a $P[i \dots j]$, assurdo.

Valore della soluzione ottima

Definizione

Sia $DP[i][j]$ il minimo numero di moltiplicazioni scalari necessarie per calcolare il prodotto $A[i \dots j]$

Valore della soluzione ottima

Definizione

Sia $DP[i][j]$ il minimo numero di moltiplicazioni scalari necessarie per calcolare il prodotto $A[i \dots j]$

- **Caso base:** $i = j$. Allora $DP[i][j] = 0$
- **Passo ricorsivo:** $i < j$. Esiste una parentesizzazione ottima

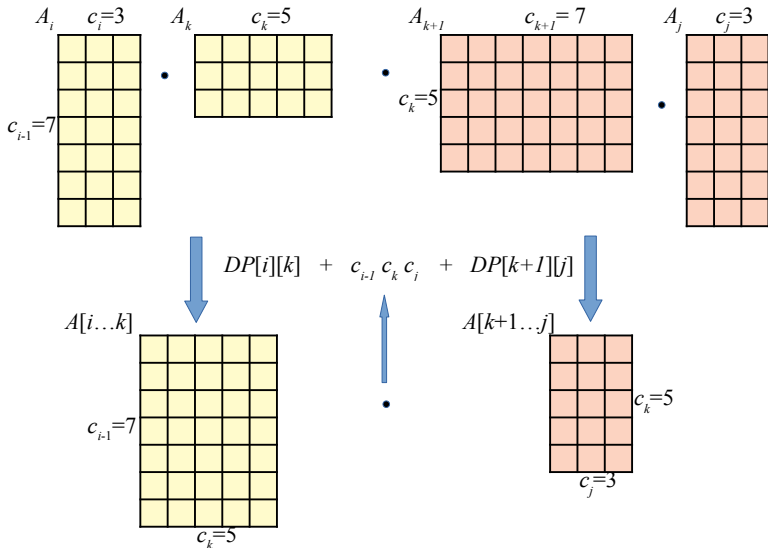
$$P[i \dots j] = P[i \dots k] \cdot P[k + 1 \dots j]$$

Sfruttando la ricorsione:

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

- $c_{i-1} \cdot c_k \cdot c_j$ è il costo per moltiplicare
 - la matrice $A_i \dots A_k$: c_{i-1} righe, c_k colonne
 - la matrice $A_{k+1} \dots A_j$: c_k righe, c_j colonne

Valore della soluzione ottima



Valore della soluzione ottima

Ma qual è il valore di k ?

- Non lo conosciamo....
- ... ma possiamo provarli tutti!
- k può assumere valori fra i e $j - 1$

Formula finale

$$DP[i][j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{ DP[i][k] + DP[k+1][j] + c_{i-1} \cdot c_k \cdot c_j \} & i < j \end{cases}$$

Esempio

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

$$\begin{aligned}
 DP[1][2] &= \min_{1 \leq k < 2} \{ DP[1][k] + DP[k+1][2] + c_0 c_k c_2 \} \\
 &= DP[1][1] + DP[2][2] + c_0 c_1 c_2 \\
 &= c_0 c_1 c_2
 \end{aligned}$$

Esempio

$i \setminus j$	1	2	3	4	5	6
1	0					
2	-	0		●		
3	-	-	0			
4	-	-	-	0		
5	-	-	-	-	0	
6	-	-	-	-	-	0

$$\begin{aligned}
 DP[2][4] &= \min_{2 \leq k < 4} \{ DP[2][k] + DP[k+1][4] + c_1 c_k c_4 \} \\
 &= \min \{ DP[2][2] + DP[3][4] + c_1 c_2 c_4, \\
 &\quad DP[2][3] + DP[4][4] + c_1 c_3 c_4 \}
 \end{aligned}$$

Esempio

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

$$\begin{aligned}
 DP[2][5] &= \min_{2 \leq k < 5} \{ DP[2][k] + DP[k+1][5] + c_1 c_k c_5 \} \\
 &= \min \{ DP[2][2] + DP[3][5] + c_1 c_2 c_5, \\
 &\quad DP[2][3] + DP[4][5] + c_1 c_3 c_5, \\
 &\quad DP[2][4] + DP[5][5] + c_1 c_4 c_5 \}
 \end{aligned}$$

Esempio

$i \setminus j$	1	2	3	4	5	6
1	0				●	
2	-	0				
3	-	-	0			
4	-	-	-	0		
5	-	-	-	-	0	
6	-	-	-	-	-	0

$$\begin{aligned}
 DP[1][5] &= \min_{1 \leq k < 5} \{ DP[1][k] + DP[k+1][5] + c_0 c_k c_5 \} \\
 &= \min \{ DP[1][1] + DP[2][5] + c_0 c_1 c_5, \\
 &\quad DP[1][2] + DP[3][5] + c_0 c_2 c_5, \\
 &\quad DP[1][3] + DP[4][5] + c_0 c_3 c_5, \\
 &\quad DP[1][4] + DP[5][5] + c_0 c_4 c_5 \}
 \end{aligned}$$

Esempio

$i \setminus j$	1	2	3	4	5	6
1	0					⊙
2	-	0				
3	-	-	0			
4	-	-	-	0		
5	-	-	-	-	0	
6	-	-	-	-	-	0

$$\begin{aligned}
 DP[1][6] &= \min_{1 \leq k < 6} \{ && DP[1][k] + DP[k+1][6] + c_0 c_k c_6 \} \\
 &= \min \{ && DP[1][1] + DP[2][6] + c_0 c_1 c_6, \\
 &&& DP[1][2] + DP[3][6] + c_0 c_2 c_6, \\
 &&& DP[1][3] + DP[4][6] + c_0 c_3 c_6, \\
 &&& DP[1][4] + DP[5][6] + c_0 c_4 c_6, \\
 &&& DP[1][5] + DP[6][6] + c_0 c_5 c_6 \}
 \end{aligned}$$

Dalla formula al codice

Input

- Un vettore $c[0 \dots n]$ contenente le dimensioni delle matrici
 - $c[0]$ è il numero di righe della prima matrice
 - $c[i - 1]$ è il numero di righe della matrice A_i
 - $c[i]$ è il numero di colonne della matrice A_i
- Due indici i, j che rappresentano l'intervallo di matrici da moltiplicare

Output

Il numero di moltiplicazioni scalari per calcolare il prodotto delle matrici comprese fra gli indici i e j

Approccio ricorsivo

```

int recPar(int[] c, int i, int j)


---


if i == j then
  | return 0
else
  | minSoFar =  $+\infty$ 
  | for int k = i to j - 1 do
  | | int val = recPar(c, i, k) + recPar(c, k + 1, j) + c[i - 1] · c[k] · c[j]
  | | if val < minSoFar then
  | | | minSoFar = val
  | return minSoFar

```

Complessità?

Valutazione

Alcune riflessioni

- La soluzione ricorsiva top-down è $\Omega(2^n)$
- Non è poi migliore dell'approccio basato su forza bruta!
- Il problema è che molti sottoproblemi vengono risolti più volte
- Il numero di sottoproblemi è $\frac{n(n+1)}{2}$

Versione bottom-up

Tabelle programmazione dinamica

Due matrici DP , $last$ di dimensione $n \times n$ tali che:

- $DP[i][j]$ contiene il numero di moltiplicazioni scalari necessarie per moltiplicare le matrici $A[i \dots j]$
- $last[i][j]$ contiene il valore k dell'ultimo prodotto che minimizza il costo per il sottoproblema

Versione bottom-up

```

computePar(int[] c, int n)


---


int[][] DP = new int[1...n][1...n]
int[][] last = new int[1...n][1...n]
for i = 1 to n do                                     % Fill main diagonal
  DP[i][i] = 0
for h = 2 to n do                                     % h: diagonal index
  for i = 1 to n - h + 1 do                             % i: row
    int j = i + h - 1                                   % j: column
    DP[i][j] = +∞
    for k = i to j - 1 do                               % k: last product
      int temp = DP[i][k] + DP[k + 1][j] + c[i - 1] · c[k] · c[j]
      if temp < DP[i][j] then
        DP[i][j] = temp
        last[i][j] = k
return DP[1][n]

```

<i>DP</i>	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>i</i>	<i>c</i> [<i>i</i>]
0	7
1	8
2	4
3	2
4	3
5	5
6	6

$$\begin{aligned}
 DP[1][4] &= \min_{1 \leq k < 4} \{ DP[1][k] + DP[k+1][4] + c_0 \cdot c_k \cdot c_4 \} \\
 &= \min \left\{ \begin{array}{l} DP[1][1] + DP[2][4] + c_0 \cdot c_1 \cdot c_4, \\ DP[1][2] + DP[3][4] + c_0 \cdot c_2 \cdot c_4, \\ DP[1][3] + DP[4][4] + c_0 \cdot c_3 \cdot c_4 \end{array} \right\} \\
 &= \min \left\{ \begin{array}{l} 0 + 112 + 7 \cdot 8 \cdot 3, \\ 224 + 24 + 7 \cdot 4 \cdot 3, \\ 176 + 0 + 7 \cdot 2 \cdot 3 \end{array} \right\} \\
 &= \min \{ 280, 332, 218 \}
 \end{aligned}$$

<i>last</i>	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>i</i>	<i>c</i> [<i>i</i>]
0	7
1	8
2	4
3	2
4	3
5	5
6	6

$$\begin{aligned}
 DP[1][4] &= \min_{1 \leq k < 4} \{ DP[1][k] + DP[k+1][4] + c_0 \cdot c_k \cdot c_4 \} \\
 &= \min \{ DP[1][1] + DP[2][4] + c_0 \cdot c_1 \cdot c_4, \\
 &\quad \{ DP[1][2] + DP[3][4] + c_0 \cdot c_2 \cdot c_4, \\
 &\quad \{ DP[1][3] + DP[4][4] + c_0 \cdot c_3 \cdot c_4 \} \\
 &= \min \{ 0 + 112 + 7 \cdot 8 \cdot 3, \\
 &\quad \{ 224 + 24 + 7 \cdot 4 \cdot 3, \\
 &\quad \{ 176 + 0 + 7 \cdot 2 \cdot 3 \} \\
 &= \min \{ 280, 332, \mathbf{218} \}
 \end{aligned}$$

Parentesizzazione ottima

Considerazioni

- Il costo computazionale è $O(n^3)$, in quanto ogni cella richiede tempo $O(n)$ per essere riempita
- Il costo della funzione si trova nella posizione $DP[1][n]$
- È anche necessario mostrare la soluzione trovata
- Per questo motivo abbiamo registrato informazioni sulla soluzione nella matrice *last*

Ricostruzione della soluzione – Stampa

```
computePar(int[] c, int n)
```

```
[...]
```

```
printPar(last, 1, n)
```

```
printPar(int[][] last, int i, int j)
```

```
if i == j then
```

```
  | print "A["; print i; print "]"
```

```
else
```

```
  | print "("; stampaPar(last, i, last[i][j]); print ".";
  | stampaPar(last, last[i][j] + 1, j); print ")"
```

Ricostruzione della soluzione – Calcolo effettivo

```
int[][] multiply(matrix[] A, int[][] last, int i, int j)
```

```
if i == j then
```

```
  | return A[i]
```

```
else
```

```
  | int[][] X = multiply(A, last, i, last[i][j])
```

```
  | int[][] Y = multiply(A, last, last[i][j] + 1, j)
```

```
  | return matrix-multiplication(X, Y)
```

Esempio

$$A[1\dots 6] = A[1\dots 3] \cdot A[4\dots 6]$$

$$A[1\dots 3] = A_1 \cdot A[2\dots 3]$$

$$A[4\dots 6] = A[4\dots 5] \cdot A_6$$

$$A[2\dots 3] = A_2 \cdot A_3$$

$$A[4\dots 5] = A_4 \cdot A_5$$

<i>last</i>	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

Risultato finale

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

Prodotto di catena di matrici

Take-home message – prendi e porta a casa

A volte, bisogna fare attenzione a come riempire la tabella - non è detto che riempire una riga dopo l'altra sia possibile.

Algoritmi più efficienti

- Hu and Shing, 1982 – Complessità $O(n \log n)$
- Wang, Zhu and Tian, 2013 – Complessità $O(n \log m)$, dove m è il numero di "minimi locali" nel vettore delle dimensioni.

Sommario

- 1 String matching approssimato
- 2 Prodotto di catena di matrici
- 3 Insieme indipendente di intervalli pesati

Insieme indipendente di intervalli pesati – Introduzione

Input

Siano dati n intervalli distinti $[a_1, b_1[, \dots, [a_n, b_n[$ della retta reale, aperti a destra, dove all'intervallo i è associato un profitto $w_i, 1 \leq i \leq n$.

Intervalli disgiunti

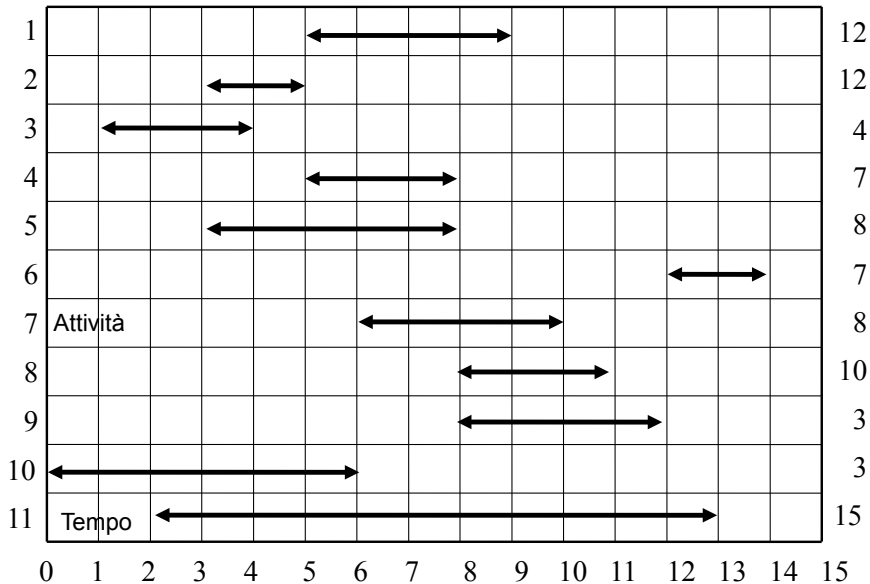
Due intervalli i e j si dicono **disgiunti** se: $b_j \leq a_i$ oppure $b_i \leq a_j$

Problema

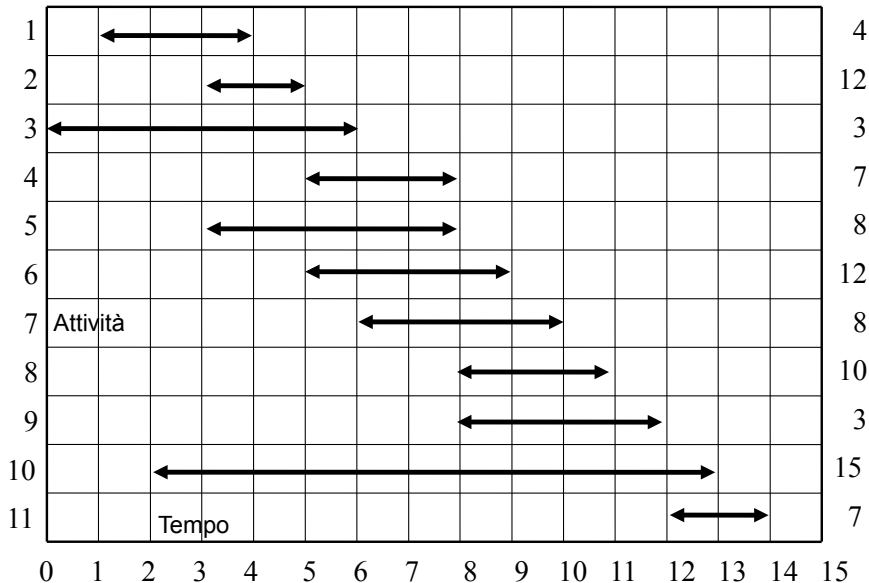
Trovare un **insieme indipendente di peso massimo**, ovvero un sottoinsieme di intervalli disgiunti tra loro tale che la somma dei loro profitti sia la più grande possibile.

- Esempio: prenotazione di una sala conferenza

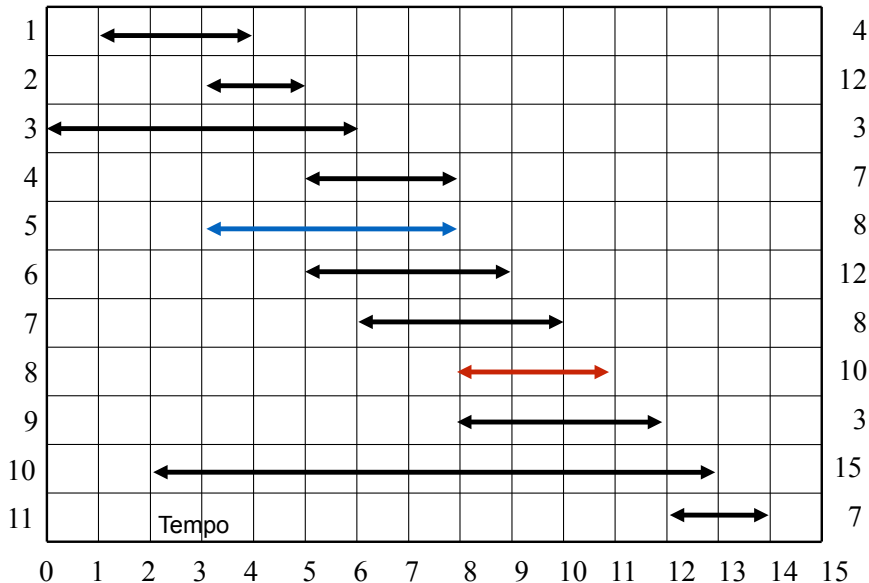
Esempio



Esempio – Ordinamento per estremi di fine non-decrescenti



Esempio – Ordinamento per estremi di fine non-decrescenti



Pre-elaborazione

Per usare PD, è necessario effettuare una pre-elaborazione: ordinare gli intervalli per estremi finali non decrescenti

$$b_1 \leq b_2 \leq \dots \leq b_n$$

Profitto massimo (Versione 1)

$DP[i]$ contiene il profitto massimo ottenibile con i primi i intervalli

$$DP[i] = \begin{cases} 0 & i = 0 \\ \max(DP[i-1], \max\{DP[j] + w_i : j < i \wedge b_j \leq a_i\}) & i > 0 \end{cases}$$

Costo computazionale?

Pre-elaborazione

Per usare PD, è necessario effettuare una pre-elaborazione: ordinare gli intervalli per estremi finali non decrescenti

$$b_1 \leq b_2 \leq \dots \leq b_n$$

Profitto massimo (Versione 1)

$DP[i]$ contiene il profitto massimo ottenibile con i primi i intervalli

$$DP[i] = \begin{cases} 0 & i = 0 \\ \max(DP[i-1], \max\{DP[j] + w_i : j < i \wedge b_j \leq a_i\}) & i > 0 \end{cases}$$

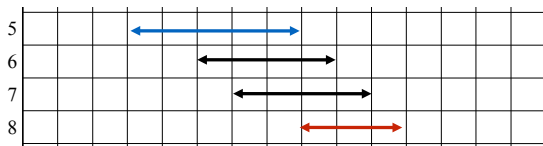
Costo computazionale?

$$O(n^2)$$

Pre-elaborazione

Una seconda possibile pre-elaborazione consiste nel pre-calcolare il predecessore $pred_i = j$ di i , dove:

- $j < i$ è il massimo indice tale che $b_j \leq a_i$
- se non esiste tale indice, $pred_i = 0$.



Profitto massimo (Versione 2)

$$DP[i] = \begin{cases} 0 & i = 0 \\ \max(DP[i-1], DP[pred_i] + w_i) & i > 0 \end{cases}$$

Pre-elaborazione - calcolo predecessori

```
int[] computePredecessor(int[] a, int[] a, int n)
```

```
int[] pred = new int[0...n]
```

```
pred[0] = 0
```

```
for i = 1 to n do
```

```

┌   j = i - 1
├   while j > 0 and b[j] > a[i] do
├   │   j = j - 1
└   pred[i] = j
```

```
return pred
```

Quanto costa pre-calcolare i predecessori?

Pre-elaborazione - calcolo predecessori

```
int[] computePredecessor(int[] a, int[] a, int n)
```

```
int[] pred = new int[0...n]
```

```
pred[0] = 0
```

```
for i = 1 to n do
```

```

┌   j = i - 1
├   while j > 0 and b[j] > a[i] do
├   │   j = j - 1
└   │   pred[i] = j
```

```
return pred
```

Quanto costa pre-calcolare i predecessori?

$O(n^2)$

Si può fare meglio di così?

Pre-elaborazione - calcolo predecessori

```
int[] computePredecessor(int[] a, int[] a, int n)
```

```
int[] pred = new int[0...n]
```

```
pred[0] = 0
```

```
for i = 1 to n do
```

```
  j = i - 1
```

```
  while j > 0 and b[j] > a[i] do
```

```
    j = j - 1
```

```
  pred[i] = j
```

```
return pred
```

Quanto costa pre-calcolare i predecessori?

$O(n^2)$

Si può fare meglio di così?

Sì! $O(n \log n)$

Alcune note

- Gli intervalli vanno ordinati per tempo non decrescente di fine
- Eventuali intervalli con lo stesso tempo di fine possono essere ordinati in qualunque modo
- Questo perché ogni valore $DP[i]$ rappresenta il massimo profitto ottenibile con i primi i intervalli
- È quindi possibile escludere l'intervallo i -esimo se sceglierne uno precedente j (ma con lo stesso tempo di fine $b[i] = b[j]$) ha un valore $DP[j]$ più alto

Versione completa

```

SET maxinterval(int[] a, int[] b, int[] w, int n)


---


{ ordina gli intervalli per estremi di fine crescenti }
int[] pred = computePredecessor(a, b, n)
int[] DP = new int[0...n]
DP[0] = 0
for i = 1 to n do
     $DP[i] = \max(DP[i - 1], w[i] + DP[pred[i]])$ 
i = n
SET S = Set()
while i > 0 do
    if  $DP[i - 1] > w[i] + DP[pred[i]]$  then
        | i = i - 1
    else
        | S.insert(i)
        | i = pred[i]
return S

```

Costo computazionale

Costo computazionale

- Ordinamento intervalli: $O(n \log n)$
- Calcolo predecessori: $O(n \log n)$
- Riempimento tabella *DP*: $O(n)$
- Ricostruzione soluzione: $O(n)$
- Algoritmo completo: $O(n \log n)$

Esercizio

Scrivere una funzione di calcolo predecessori in tempo $O(n \log n)$

Insieme indipendente di intervalli pesati – Conclusioni

Take-home message – prendi e porta a casa

Talvolta, può essere necessario pre-processare l'input per poter applicare nella maniera più efficiente la programmazione dinamica

Per concludere

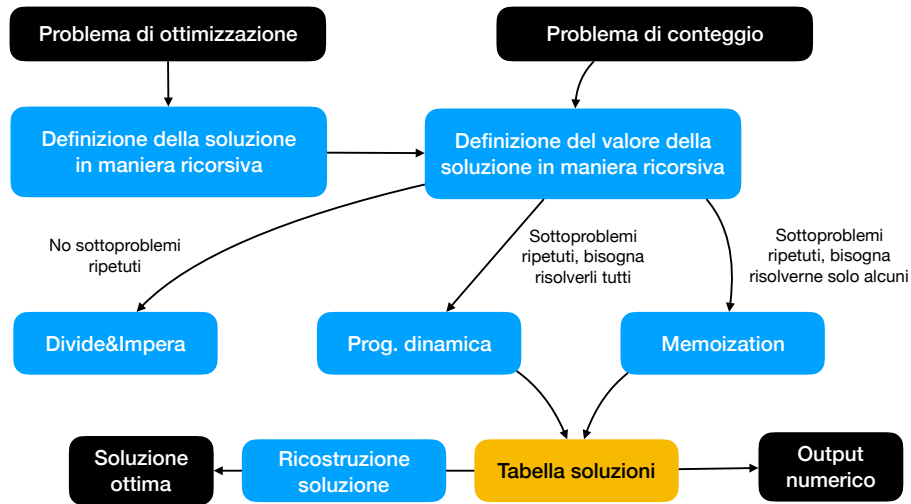
Una lezione ancora più importante

La programmazione dinamica non è la soluzione di tutti i vostri problemi. Esistono altre tecniche che possono fare "meglio di così". Inoltre, è possibile che soluzioni ad-hoc possano essere migliori

Esempi

- **Moltiplicazione di matrici**: esiste soluzione $O(n \log n)$
- **Longest common subsequence**: può essere risolto in tempo $O(mn / \log n)$ (con alfabeto limitato)
- **Four Russians algorithm**: tecnica generale che può essere applicata a vari problemi su matrice con alfabeto limitato $O(n^2 / \log n)$
 - Edit distance
 - Transitive closure, ...

Approccio generale



Riassunto: programmazione dinamica / memoization

Fasi

- Caratterizzare la **struttura** di una soluzione ottima
- Dimostrare che la soluzione gode di **sottostruttura ottima**
- Definire ricorsivamente il **valore** di una soluzione ottima
- Calcolare il **valore** di una soluzione ottima "bottom-up" (prog. dinamica) / "top-down" (memoization)
- **Ricostruzione** di una soluzione ottima