

Programmazione dinamica

/ 23-03

È un paradigma (schema) algoritmico
Un altro schema noto è il 'divide et impera'

E' utile quando si applicano dove la soluzione di un problema si ottiene combinando soluzioni di sotto problemi più piccoli

Lo schema e' induuttivo: BASE + PASSO INDUTTIVO

Numeri di Fibonacci; 1, 1, 2, 3, 5 ... il numero e' risultato delle somme dei due precedenti

$$fib_0 = 1$$

$$fib_1 = 1$$

$$\forall n > 1 \quad fib_n = fib_{n-1} + fib_{n-2}$$

RICORSIONE:

```
fb(n) = if (n <= 1)
        return 1
        return fb(n-1) + fb(n-2)
```

complessità:

esponenziale

Similmente alle torri di Hanói $T(n) = T(n-1) + T(n-2)$

Un algoritmo più semplice:

$fib(n)$

ITERAZIONE

array di dim m FIB

$$FIB[0] = 1$$

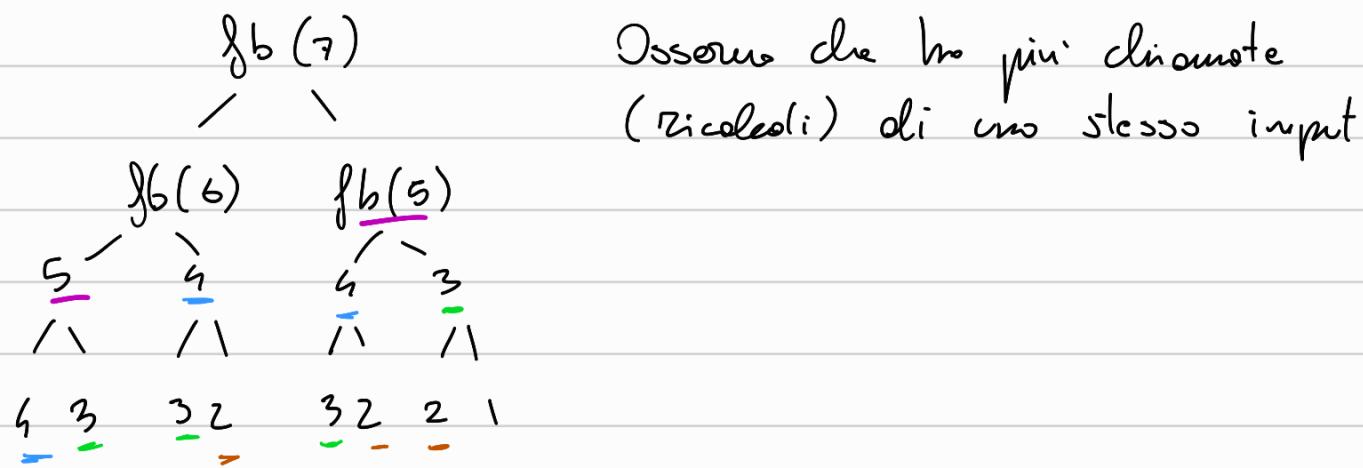
$$FIB[1] = 1$$

for each $i = 2$ to n

$$FIB[i] = FIB[i-1] + FIB[i-2]$$

costo lineare

Il punto chiave è capire che tanti sottoproblemi vengono ricalcolati più volte. Ecco perché il costo esponenziale:



La programmazione (= pianificazione) dinamica è quindi:

- basata su induzione
- basata su bottom-up: parto dai problemi base fino al problema top (il caso esponenziale è un Top-bottom)
(piccolo)
(grande)
- vantaggio: si ha se un sotto problema serve più volte
(conservo una volta sola il risultato)
- svantaggio: tutti i sottoproblemi vanno calcolati nel caso di bottom-up (devo solvere necessariamente anche gli intermedi)

Longest Common Subsequence (LCS) / 04-04

Dato due sequenze trovare una sottosequenza comune a entrambe di lunghezza massima (rispetto l'ordine ma gli elementi non sono necessariamente contiguous)

es AGC CG GAT CG A GT
 TC AG TA CG T TA

sottosep.: AGC GT A
(oppure AGT CGA)

L'algoritmo non restituisce le sottosequenze, bensì la posizione degli elementi in una coppia relativa a un elemento di ogni sequenza

AGC CG GAT CG A GT
TC AG TA CG T TA
[(0,2), (1,3), (2,6) ...]

Vantaggi:

- non duplico informazioni
(non alloco nuove sequenze)
- ho più informazione
(conosco le posizioni degli elementi)

Risultato: liste di coppie da cui dedursi le sottosequenze

Algoritmo brute force

- genero tutte le sottosequenze di s_1 e confronto se è anche a s_2 tenendo conto di quella di lunghezza max

Note: le sottosequenze ha ordine, quindi la quantità di sottosep. di s_1 non sono le permutazioni, bensì 2^m
(ocelgo se inserire o meno l'elemento corrente)

Per controllare se $\in S_2$ servono n passi (a lunghezza s_2):
controllare se l'elemento di s_1 e' in S_2

Totale $n \cdot 2^m$ = esponenziale

Algoritmo di programmazione dinamica
Formalizzazione induttiva

Base: siano $X[1...m]$ e $Y[1...n]$ le due sequenze

sottoproblema $LCS(i, j)$ sottoseq di 1 mox
tra $X[1...i]$ e $Y[1...j]$

$$\rightarrow LCS(0, j) = [] \quad \text{per } 0 \leq j \leq n$$
$$\rightarrow LCS(i, 0) = [] \quad \text{per } 0 \leq i \leq m$$

↓
caso el.

perche' non ho sottoseq. comune

P.J.:

caso 1 - esaminiamo gli ultimi due, suppose $X[i] = Y[j]$

Dunque se $LCS(:i-1, :j-1) = [(i_0, j_0), (i_1, j_1) \dots (i_n, j_n)]$

allora $LCS(i, j) = [(i_0, j_0), (i_1, j_1) \dots (i_n, j_n), (i, j)]$

caso 2: $X[i] \neq Y[j]$

le coppie (i, j) non sono inserite in una sottoseq comune
oppure un certo $i' \neq j'$: (i, j') , (i', j)

Quindi sono sottoseq comuni di $X[1 \dots i-1]$ e $Y[1 \dots j-1]$

o $X[1 \dots i]$ e $Y[1 \dots j-1]$

Quindi:

$$\begin{aligned} \text{se } x(i) = y(i) \quad & LCS(i, j) = LCS(i-1, j-1) + 1 \\ \text{se } x(i) \neq y(i) \quad & max(LCS(i-1, j), LCS(i, j-1)) \\ & \quad \swarrow \quad \nwarrow \\ & \quad \text{2 diverse} \end{aligned}$$

Alternativamente potrei restituire il numero corrispondente alla lunghezza
de forme le sottoseq.

Le dim di un sottoproblema e' la coppia dei prefissi considerati

→ vogliamo dim sottoproblemi < dim problema di pertenza

Come dire se $(i_1, j_1) < (i_2, j_2)$?

mi basta che $i_1 < i_2$ e $j_1 \leq j_2$

o $i_1 \leq i_2 \text{ e } j_1 < j_2$

Definizione induttiva = eleg di dividere et impiere (ricorsivo)

Relazione di ricorrenza (caso peggiore: max di due chiamate ricorsive)

$$T(n, m) = T(n-1, m) + T(n, m-1) + 1$$

Considero $\kappa = n+m$

$$T(\kappa) = 2T(\kappa-1) + 1 \quad \text{relazione di ricorrenza delle Torri di Hanoi}$$

(definizione esponenziale)

Det: i così sopra (del passo induttivo)

e partire dal problema grande ho 3 sottoproblemi:

e partire dal sottoproblema $LCS(i, j)$ proli:

sottoproblemi di dim maggiore posso calcolare?

$$\begin{cases} LCS(i+1, j+1) \\ LCS(i+1, j) \\ LCS(i, j+1) \end{cases}$$

Guardando presto secondo cosa un sotto problema viene quindi usato più volte per calcolare quelli di dimensione maggiore
 → posso usare la programmazione dinamica

Alg di programmazione dinamica

• matrice di m righe e n colonne

- primo righe - colonne per le sequenze vuote
- → ricopre riga > riga (\Rightarrow riga > colonna)
- La casella $LCS(m, n)$ contiene la soluzione

	A	T	C	B	A	B
A	0	0	0	0	0	0
B	0	0↑	0↑	0↑	↖ 1	↖ 1
A	0	↖ 1	↖ 1	↖ 1	=↑ 1	↖ 2
C	0	↑ 1	↑ 1	↖ 2	↑ 2	↑ 2
A	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3
T	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3
B	0	↑ 1	↑ 2	↑ 2	↖ 3	↖ 4
A	0	↑ 1	↑ 2	↑ 2	↑ 3	↖ 4

m: modo per sottosequenze

A, C, A, B

C, A, B

A, B

B

posso ricostruire anche da qui,
 ottengo un'altra sottosequenza

↳ LCS delle seq intere

Ricostrovo la sequenza seguendo le frecce, salvando solo gli elementi che presentano stessa valori nello casello. Ottengo A, C, A, B

Da una casella devo uscire altre 3:



(3 sotto problemi)

L'algorithm utilizza una matrice che occupa spazio $O(nm)$

```

for (i = 0; i <= m; i++) L[i,0] = 0
for (j = 0; j <= n; j++) L[0,j] = 0
for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++)
        if (X[i] = Y[j])
            L[i,j] = L[i-1,j-1] + 1; R[i,j] = ↗
        else if (L[i,j-1] > L[i-1,j])
            L[i,j] = L[i,j-1]; R[i,j] = ←
        else L[i,j] = L[i-1,j]; R[i,j] = ↑
    
```

→ prime riga e colonna (sep. nula)

Cost: quadrato

$\Theta(m \cdot n)$

Come precedentemente detto, non c'è obiettivo ricostituire la sottosequenza

Si puo' ulteriormente ottimizzare lo spazio occupato nel caso in cui non mi interessa (e interessi) solo la lunghezza (lungo perito della matrice, mi riconduca allo lineare).

Algoritmo di Floyd e Warshall - su grafi

Trovare cammino minimo fra tutte le coppie di nodi dato un grafo orientato pesato

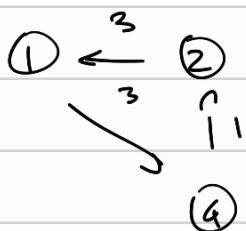
↳ e non solo da una sorgente

→ sono permessi pesi negativi ma non ci si di costo negativo

(avrei ad ogni ciclo un cammino che costa meno, non avrei un cammino minimo fissato!).

Nota: dato un cammino minimo fra due nodi costituito da nodi intermedi, anche i "parti" di cammino che coinvolgono i nodi intermedi sono e

loro volta cammini minimi fra i nodi adiacenti



cammino minimo 4, 1

è 4 → 1 → 2, costo 7

ma allo stesso tempo 4, 2 è cammino minimo fra 4 e 2 con costo 1

Questo da l'intuizione dei pesi = sotto problemi:

Chiamiamo **κ -vincolato** un cammino che possa solo per i nodi in $1 \dots n$ (esclusi gli estremi) per $\kappa \leq n$ (n tot nodi). Numero i nodi $1 \dots n$

Sotto problema: $d^\kappa(x, y) =$ distanza κ -vincolata tra x e y cioè la lunghezza di un cammino κ -vincolato (o se il cammino \emptyset)

Caso base:

$$d^0(x, y) = \begin{cases} 0 & \text{se } x = y \\ c_{x,y} & \text{se } x \neq y \text{ col } \exists \text{ arco } (x,y) \\ \infty & \text{altrimenti} \end{cases}$$

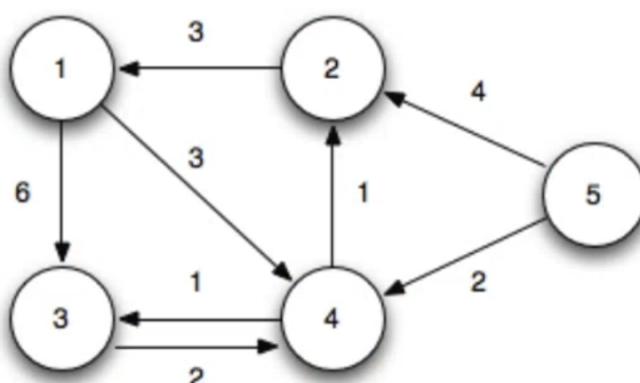
sotto problema

$$P.J.: d^\kappa(x, y) = \min \left\{ \underline{d^{\kappa-1}(x, y)}, \underline{d^{\kappa-1}(x, \kappa)} + \underline{d^{\kappa-1}(\kappa, y)} \right\}$$

Dato un cammino min (semplice) κ -vincolato da x a y ho 2 casi:

- non posso per κ , quindi è camminino $\kappa-1$ -vincolato
 - posso per κ , quindi poiché semplice è composto da un cammino $\kappa-1$ -vincolato da x a κ e da κ a y
- excluendo gli estremi
(fra cui il κ -esimo)

Per semplice si intende che posso per κ una volta sola (no cicli, i cammini con nodi ripetuti non sono minimi)



- $d^0(5,3) = \infty$ (non ho cammino senza altri nodi fra 5 e 3)
- $d^1(5,2) = 4$
- $d^1(5,3) = \infty$ (può posso usare solo il nodo numerato 1 = κ)
- $d^2(5,3) = 13$ ($\kappa=?$, posso usare il nodo 1 e il nodo 2 = κ)
- $d^3(5,3) = 13$ non cambia nulla
- $d^4(5,3) = 3$ migliore ($\kappa=4$ posso usare i nodi fino a κ)

Quando aggiunge un nodo (d^0 , d^1 e d^2) mi raffiguro i due S , i comuni \wedge tot sono: quelli precedenti che non passano per z + comuni che vengono da S se z non è z (altrimenti ci ripasso) + comuni da $z = 1$

Algoritmo di programmazione dinamica

Ostruisco tutte le matrici per i vari v (utilizzo n matrici)

FloydWarshall(G)

```

for each (x,y : nodi in G)
    D0[x,y] =
        0 se x=y
        cx,y se x ≠ y ed esiste arco (x,y)
        ∞ altrimenti
for (k=1; k <= n; k++)
    for each (x,y : nodi in G)
        Dk[x,y] = Dk-1[x,y] → inizializzo (non posso per v)
        if (Dk-1[x,k] + Dk-1[k,y] < Dk[x,y]) → min
            Dk[x,y] = Dk-1[x,k] + Dk-1[k,y] → se comune passante per v e' min
return Dn
```

↳ Comune
minimo

COSTO: $\Theta(n^3)$

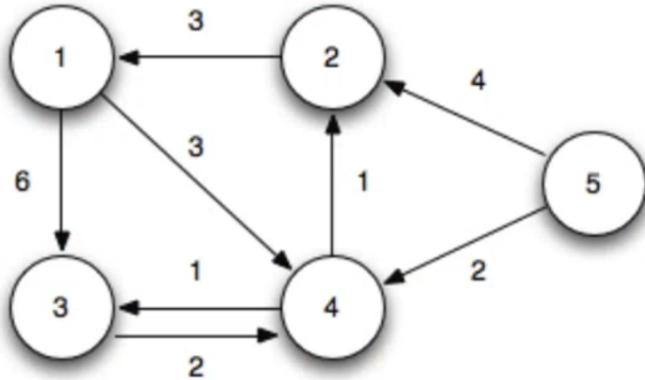
può d'essere n matrici quadrate (n^2)

cubico (ha 3 for: for su v , for su x e for su y)

spaziale

Spazialmente posso non costruire n matrici quadrate, ma posso usare un'unica matrice modificandola (temporaneamente non cambia), ovvero nell'algoritmo non ho D^k ma un'unica D . Spazio occupato n^2
 Non c'è problema nel caso i valori $D(x,v)$, $D(v,y)$ siano già aggiornati da un'iterazione e l'altra, perché se anche v diventasse intermedio (da nodo finale) aggiungeri: cicli che semplicemente non rendono il comune minimo: $x \rightarrow v$ diretta $x \rightarrow v \rightarrow x$
 $D^k(x,v) = D^{k-1}(x,v)$, cioè il minimo è non passato per v se
 dunque $D^k(x,v) = \min(D^{k-1}(x,v), D^{k-1}(v,v) + D^{k-1}_{(v,x)})$ } ripetuto dopo eventuali aggiornamenti,
 il comune min non cambia

L'algoritmo restituisce le lunghezze, se vogliamo ottenere il *comune minimo* definisco un array di pedri $\pi_{x,y}$ = null se $x = y$ o \exists com. fra x e y
 altrimenti x è predecessore di y in un comune minimo



$$\pi^0(2,3) = \text{null} \quad \text{non ha comune}$$

$$\pi^0(5,5) = \text{null}$$

$$\pi^0(5,2) = 5 \quad (x=5 \text{ predecessore})$$

$$\pi^1(2,3) = 1 \quad (\text{comune min e } 2 \rightarrow 1 \rightarrow 3, \text{ il predecessore di } 3 \text{ è } 1)$$

Formulazione induttiva

base: $\pi^0(x,y) = \begin{cases} x & \text{se } x=y \text{ ed } \exists \text{ arco } (x,y) \\ \text{null} & \text{altrimenti} \end{cases}$

f. I: $\pi^k(x,y) = \begin{cases} \pi^k(u,y) & \text{se } D[x,u] + D[u,y] < D[x,y] \\ \pi^{k-1}(x,y) & \text{altrimenti} \end{cases}$

\rightarrow se il comune non viene aggiornato
il predecessore è lo stesso di $\pi^{k-1}(x,y)$

se posso per u
(comune aggiornato),
il predecessore
è comune (k,y)

FloydWarshall(G)

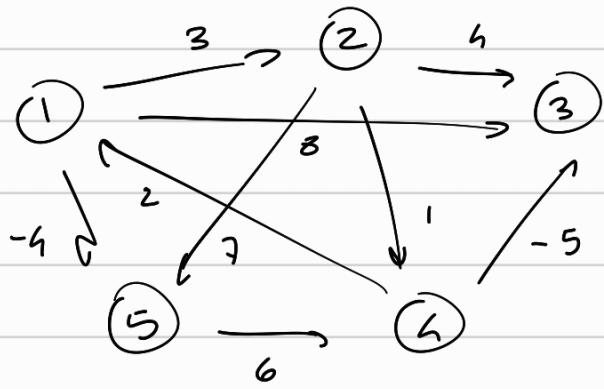
```

for each (x,y : nodi in G)
  D[x,y] =
    0 se x=y,
    cx,y se x ≠ y ed esiste arco (x,y)
    ∞ altrimenti
  Π[x,y] =
    x se x ≠ y ed esiste arco (x,y)
    null altrimenti
for (k=1; k <= n; k++)
  for each (x,y : nodi in G)
    if (D[x,k] + D[k,y] < D[x,y])
      D[x,y] = D[x,k] + D[k,y]
      Π[x,y] = Π[k,y]
return D, Π
  
```

andando nell'indietro
nell'array dei nodi
ottengo il grafo

Esempio

105-04



	1	2	3	4	5
1	0 N	3 1	8 1	∞ N	-4 1
2	∞ N	0 N	∞ N	1 2	7 2
3	∞ N	4 3	0 N	∞ N	∞ N
4	2 4	∞ N	-5 4	0 N	∞ N
5	∞ N	∞ N	6 5	6 5	0 N

costo \min_{min} = 0 N = null

comuni divetti

δ^* e π^*

	1	2	3	4	5
1	0 N	3 1	8 1	∞ N	-4 1
2	∞ N	0 N	∞ N	1 2	7 2
3	∞ N	4 3	0 N	∞ N	∞ N
4	2 4	5 1	-5 4	0 N	-2 1
5	∞ N	∞ N	6 5	6 5	0 N

qui, anche aggiungendo 1, non ho comuni fra se 1, quindi la riga non viene aggiornata

primo iterazione

aggiungo nodo $k=1$

(1^a riga e colonne restanti uguali, non aggiungo più celle 1)

idea: da 2 a 3, aggiungendo 1, cambia/meglio?

(velocemente posso sommare il costo nell'1^a colonna e nell'a 1^a riga, celle corrisp.)

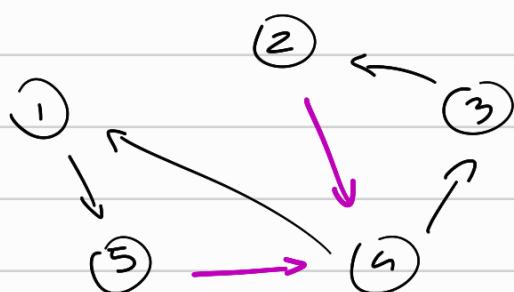
$x \rightarrow$	1	2	3	4	5	
1	0 N	1 3	-3 4	2 5	<u>-4</u> 1	
2	3 6	0 N	-4 -4	1 2	<u>-1</u> 1	
3	7 6	4 3	0 N	5 2	3 1	
4	2 6	-1 3	-5 4	0 N	-2 1	
5	8 6	5 3	1 4	6 5	0 N	

→ predecessori di 1

dopo l'ennesima
iterazione $\kappa = 5$

$$D^S \quad \pi^S$$

$$\tilde{D}(x, y) \quad \tilde{\pi}(x, y)$$



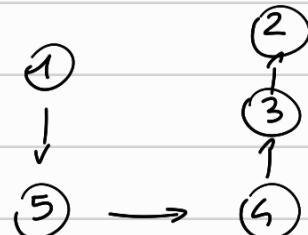
grado dei comuni

$$\text{minimi: } G_{\pi}$$

preso una coppia s, t il comun
di costo minimo da seguire
per raggiungere gli obiettivi

Fissato x ottengo l'albero dei comuni minimi da x , detto $G_{\pi, x}$

c.d. $x = 1$



G_{π} ha tutti i nodi, ha arco (z, y) se $\pi(x, y) = z$ per qualche x

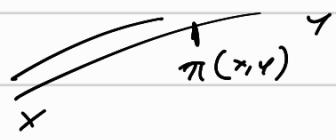
$G_{\pi, x}$ prende la parte del grafo visitata → partire da x (albero di visita)
→ i nodi sono quelli raggiungibili da x (nell'es. tutti)

Funzione shortest-path data π

sh-path(x, y) =

if ($x = y$) return x

if $\pi[x, y] = \text{null}$ "non c'e' cammino"



return sh-path($x, \pi[x, y]$) • y

\rightarrow nodo a ritorno

riprendivamente