POLITECNICO
DI TORINO

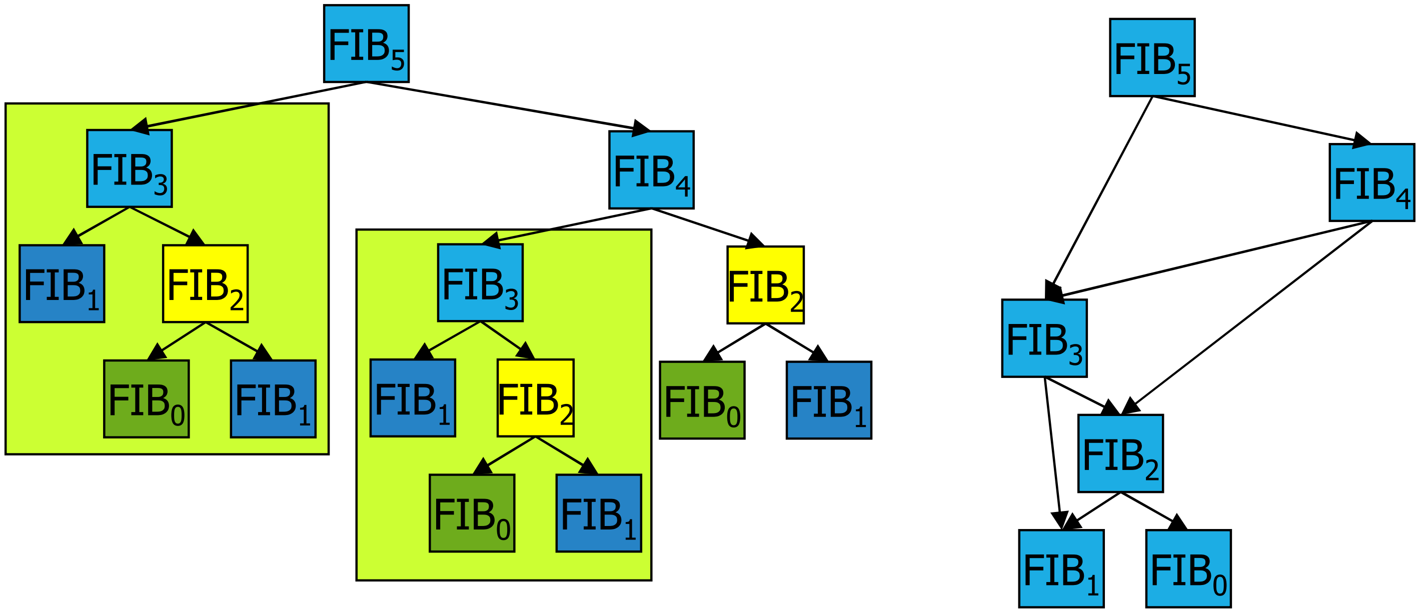
Dipartimento
di Automatica e Informatica

Il paradigma della Programmazione Dinamica

Paolo Camurati

Limiti della ricorsione

- Ipotesi di indipendenza dei sottoproblemi
- Memoria occupata



Paradigma alternativo: Programmazione Dinamica:

- memorizza le soluzioni ai sottoproblemi man mano che vengono trovate
- prima di risolvere un sottoproblema, controlla se è già stato risolto
- riusa le soluzioni ai sottoproblemi già risolti
- meglio del divide et impera per sottoproblemi condivisi

- procede:
  - bottom-up, mentre il divide et impera è top-down
  - top-down e si dice ricorsione con memorizzazione o memoization
- applicabile:
  - a problemi di ottimizzazione
  - solo se sono verificate certe condizioni
- passi:
  - verifica di applicabilità
  - soluzione ricorsiva come «ispirazione»
  - costruzione bottom-up iterativa della soluzione.

Esempio: le catene di montaggio

- Problema di ottimizzazione
- Risolvibile con:
    - i modelli del Calcolo Combinatorio
    - il paradigma divide et impera ricorsivo
    - la programmazione dinamica bottom-up
- Dall’esempio si indurrà la metodologia.

$$\left.\begin{array}{l}\text{i modelli del Calcolo Combinatorio}\\ \text{il paradigma divide et impera ricorsivo}\end{array}\right\}\ \Theta(2^n)$$

la programmazione dinamica bottom-up $\Theta(n)$

Formulazione del problema

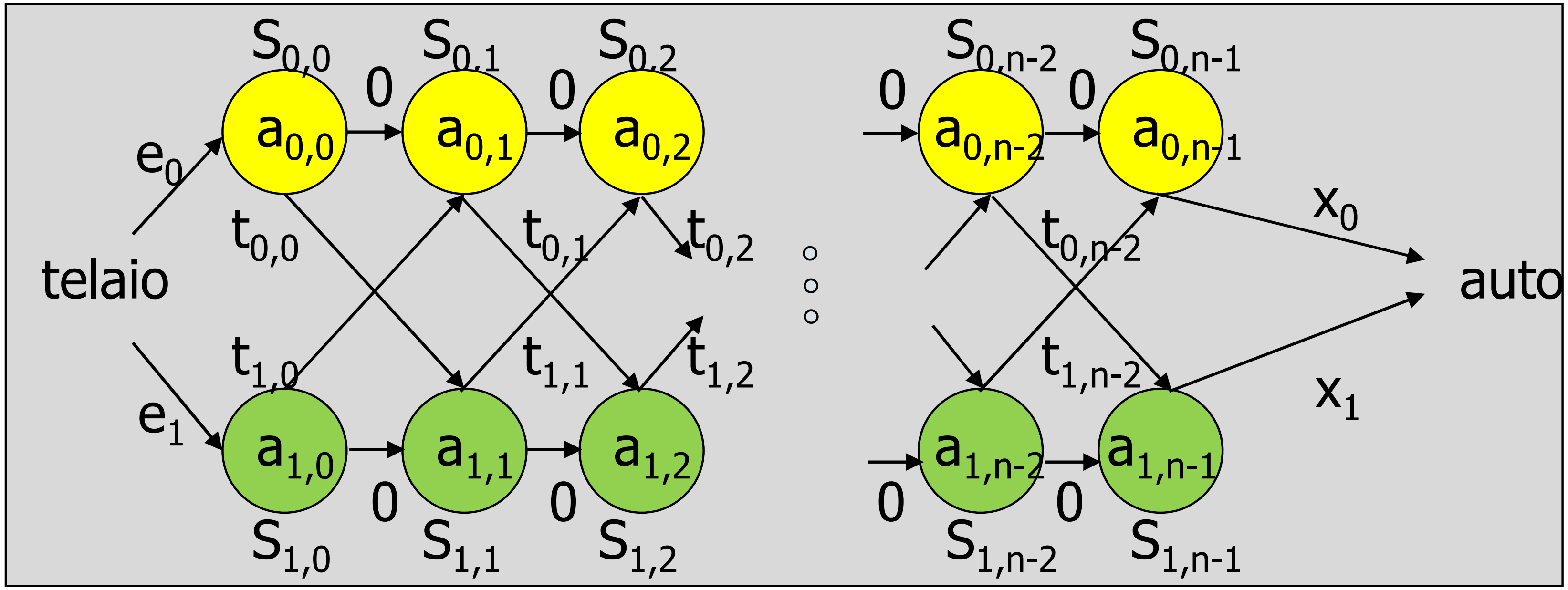
- Da telaio ⇒ ad automobile: passaggio per *n* stazioni di montaggio
- Fabbrica con:

  catena

  stazione

  - 2 catene di *n* stazioni $S_{i,j}$ ciascuna $(0 \leq i \leq 1, 0 \leq j < n)$
  - coppie di stazioni corrispondenti $S_{0,j}$ e $S_{1,j}$ svolgono la stessa funzione ma con tempi di lavorazione $a_{0,j}$ e $a_{1,j}$ diversi
  - tempo nullo per trasferimento da una stazione $S_{i,j-1}$ alla successiva nella stessa catena $S_{i,j}$

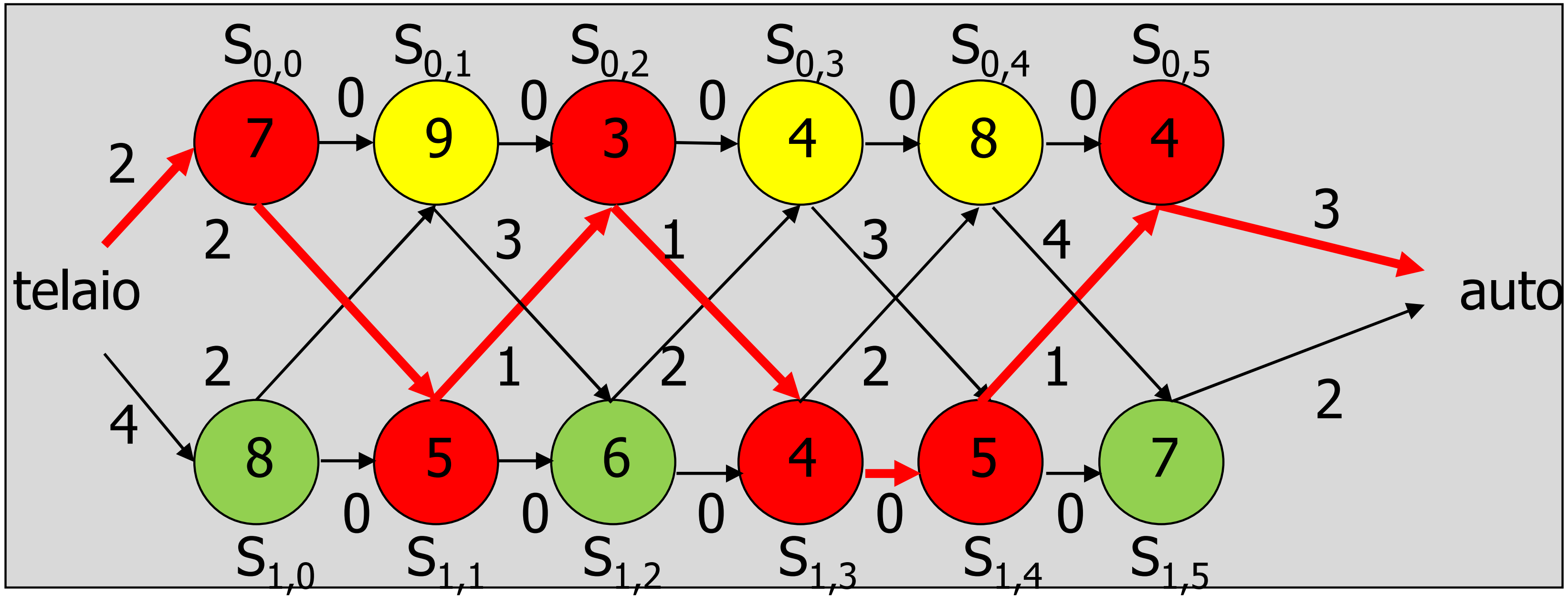
- tempo $t_{i,j-1}$ per passare da una stazione $S_{i,j-1}$ di una catena alla stazione successiva dell'altra catena $S_{(i+1)\%2,j}$
- tempi di entrata/uscita e_0/e_1 o x_0/x_1 in o da ciascuna catena



🔴 stazione usata

Scopo: costruire un’auto nel tempo minimo

Esempio:



Tempo minimo: 38

Soluzione «forza bruta»:

- modello: principio di moltiplicazione
- albero binario completo di altezza n+1
- DAG considerando lo scolo
- enumerazione dei cammini con complessità $T(n) = \Theta(2^n)$

La Programmazione Dinamica (Bellman, 1957)

- Applicata a problemi di ottimizzazione
- Passi:
  - verifica di applicabilità: caratterizzazione della **struttura** di una soluzione ottima
  - ispirazione: definizione ricorsiva del **valore** di una soluzione ottima
  - soluzione:
    - calcolo bottom-up del **valore** di una soluzione ottima
    - costruzione di una **soluzione** ottima.

tempo minimo = 38

stazioni: $S_{0,0}, S_{1,1}, S_{0,2}, S_{1,3}, S_{1,4}, S_{0,5}$

Struttura della soluzione ottima

Supponiamo di raggiungere la j-esima stazione $S_{0,j}$ della catena 0 con costo (tempo) minimo $f_0[j]$ dall'entrata fino all'uscita da essa:

- se j=0: si entra nella catena con costo e_0 e si somma il costo $a_{0,j}$ della stazione $S_{0,j}$

$$f_0[j] = e_0 + a_{0,j}$$

costo di entrata

costo della stazione $S_{0,j}$

- se $1 \leq j < n$:
  - o si proviene dalla stessa catena (stazione $S_{0,j-1}$) con costo $f_0[j-1]$, costo di trasferimento nullo e costo $a_{0,j}$ della stazione $S_{0,j}$

    $$f_0[j] = f_0[j-1] + a_{0,j}$$

    costo della stazione $S_{0,j-1}$

    costo della stazione $S_{0,j}$
  - o si proviene dall'altra catena (stazione $S_{1,j-1}$) con costo $f_1[j-1]$, costo di trasferimento $t_{1,j-1}$ e costo $a_{0,j}$ della stazione $S_{0,j}$

    $$f_0[j] = f_1[j-1] + t_{1,j-1} + a_{0,j}$$

    costo della stazione $S_{0,j}$

    costo della stazione $S_{1,j-1}$

    costo del trasferimento $t_{1,j-1}$

Ipotesi:

$f_0[j]$ *minimo && la stazione precedente appartiene alla stessa catena* ($S_{0,j-1}$)

Tesi:

il costo $f_0[j-1]$ ***deve essere minimo.***

Dimostrazione (per assurdo):

se $f_0[j-1]$ non fosse minimo, $\exists\ f'_0[j-1] < f_0[j-1]$. Allora $f'_0[j] = f'_0[j-1] + a_{0,j} < f_0[j]$ e si contraddirebbe l'ipotesi di $f_0[j]$ minimo.

Ipotesi:

$f_0[j]$ *minimo && la stazione precedente appartiene all'altra catena* ($S_{1,j-1}$)

Tesi:

il costo $\mathbf{f_1[j-1]}$ ***deve essere minimo***.

Dimostrazione (per assurdo):

se $f_1[j-1]$ non fosse minimo, $\exists\, f'_1[j-1] < f_1[j-1]$. Allora $f'_0[j] = f'_1[j-1] + t_{1,j-1} + a_{0,j} < f_0[j]$ e si contraddirebbe l'ipotesi di $f_0[j]$ minimo.

Analogamente per $S_{1,j.}$

Conclusione: la soluzione ottima del problema comporta che siano ottime le soluzioni ai suoi sottoproblemi ⇒ **sottostruttura ottima**.

La Programmazione Dinamica è applicabile solo a quei problemi di ottimizzazione che hanno una sottostruttura ottima.

Soluzione ricorsiva

Valore della soluzione ottima:

$$\text{soluzione} = \min(f_0[n-1] + x_0,\ f_1[n-1] + x_1)$$

costo di uscita dall'ultima stazione della catena 1

costo di uscita dall'ultima stazione della catena 0

costo di uscita dalla catena 0

costo di uscita dalla catena 1

$$f_0[j] = \begin{cases} f_0[0] = e_0 + a_{0,0} & j=0 \\ \min(f_0[j-1]+a_{0,j}\ ,f_1[j-1]+t_{1,j-1}+a_{0,j}\ ) & 1 \leq j < n \end{cases}$$

$$f_1[j] = \begin{cases} f_1[0] = e_1 + a_{1,0} & j=0 \\ \min(f_1[j-1]+a_{1,j}\ ,f_0[j-1]+t_{0,j-1}+a_{1,j}\ ) & 1 \leq j < n \end{cases}$$

```
int mCostR(int **a, int **t, int *e, int *x, int j, int i) {
  int ris;
  if (j==0)
    return e[i] + a[i][j];
  ris = min(
            mCostR(a,t,e,x,j-1,i)+a[i][j],
            mCostR(a,t e,x,j-1,(i+1)%2)+t[(i+1)%2][j-1]+a[i][j]
           );
  return ris;
 }

int assembly_line(int **a, int **t, int *e, int *x, int j){
  return min(
             mCostR(a,t, e, x, j, 0) + x[0],
             mCostR(a,t, e, x, j, 1) + x[1]
            );
}
```

solo calcolo del costo minimo

i: catena corrente

(i+1)%2: altra catena

01assembly_line

Limiti della soluzione ricorsiva

- complessità: $T(n) = \Theta(2^n)$

Soluzione ottima: calcolo bottom-up del valore

Strutture dati:

- tabella f[0…1, 0…n-1] per memorizzare i costi f[i, j] e identificare il costo minimo
- tabella l[0…1, 0…n] per tenere traccia della catena di montaggio in cui la stazione j-1 è usata nel percorso a costo minimo per raggiungere la stazione j

non serve per il calcolo del costo minimo, servirà per costruire la soluzione

Passi:

- iniziale: calcolare i costi (minimi per definizione) di $f_0[0]$ e $f_1[0]$
- intermedi: per ogni stazione intermedia di ciascuna catena decidere se costa meno raggiungerla dalla precedente:
  - rimanendo nella stessa catena
  - proveniendo dall'altra catena

  e calcolare il costo minimo
- finale: decidere se uscire dalla stazione n-1 esima della catena 0 o 1.

calcolo del costo minimo

```
int assembly_lineDP(int **a, int **t, int *e, int *x,
                    int **f, int **l, int n){
  int j, res;

  f[0][0] = e[0] + a[0][0];
  f[1][0] = e[1] + a[1][0];
```

passo iniziale catena 0

passo iniziale catena 1

```
for (j=1; j<n; j++) {
  if (f[0][j-1]+a[0][j]<=f[1][j-1]+t[1][j-1]+a[0][j]){
    f[0][j]=f[0][j-1]+a[0][j];
    l[0][j]=0;
  }
  else {
    f[0][j]=f[1][j-1]+t[1][j-1]+a[0][j];
    l[0][j]=1;
  }
  if (f[1][j-1]+a[1][j]<=f[0][j-1]+t[0][j-1]+a[1][j]) {
    f[1][j]=f[1][j-1]+a[1][j];
    l[1][j]=1;
  }
  else {
    f[1][j]=f[0][j-1]+t[0][j-1]+a[1][j];
    l[1][j]=0;
  }
}
```

passi intermedi

catena 0: vengo da catena 0

catena 0: vengo da catena 1

catena 1: vengo da catena 1

catena 1: vengo da catena 0

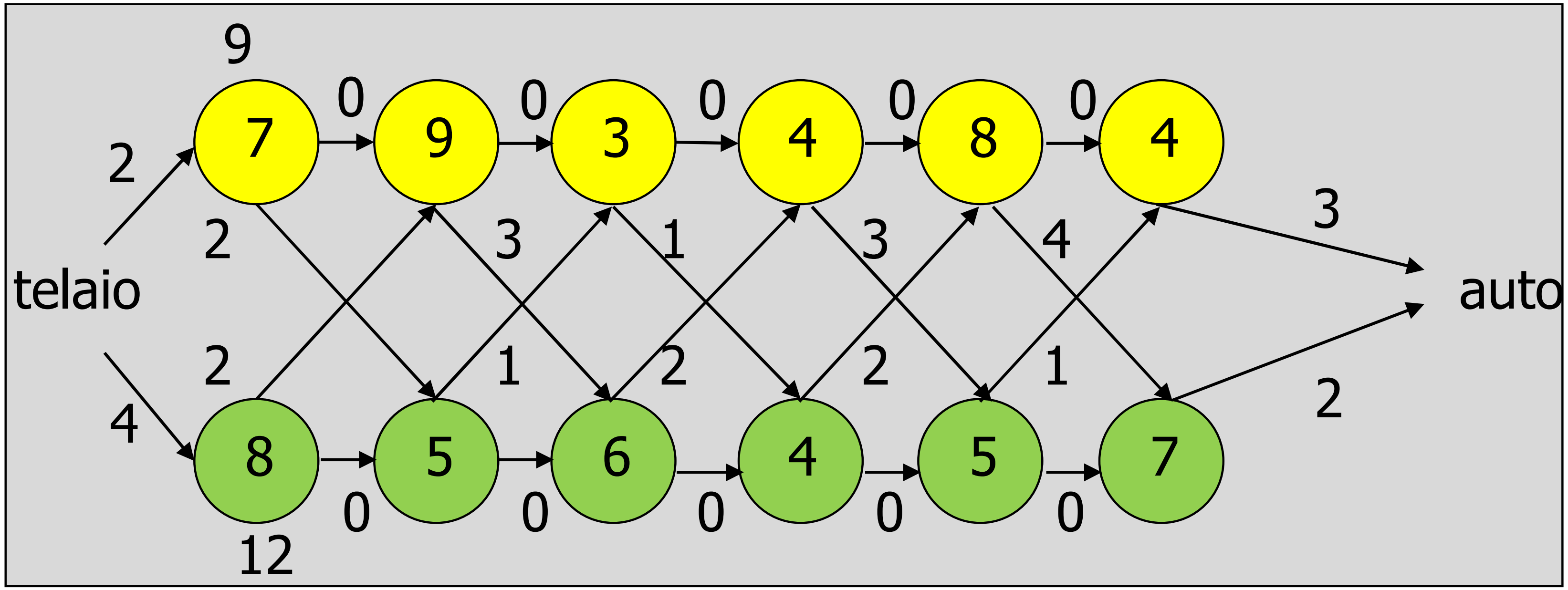
passo finale

```
if (f[0][n-1] + x[0] <= f[1][n-1] + x[1]) {
    res = f[0][n-1] + x[0];
    l[0][n] = 0; l[1][n] = 0;
  }
  else {
    res = f[1][n-1] + x[1];
    l[1][n] = 1; l[0][n] = 1;
  }
  return res;
}
```

esco dalla catena 0

esco dalla catena 1

Esempio

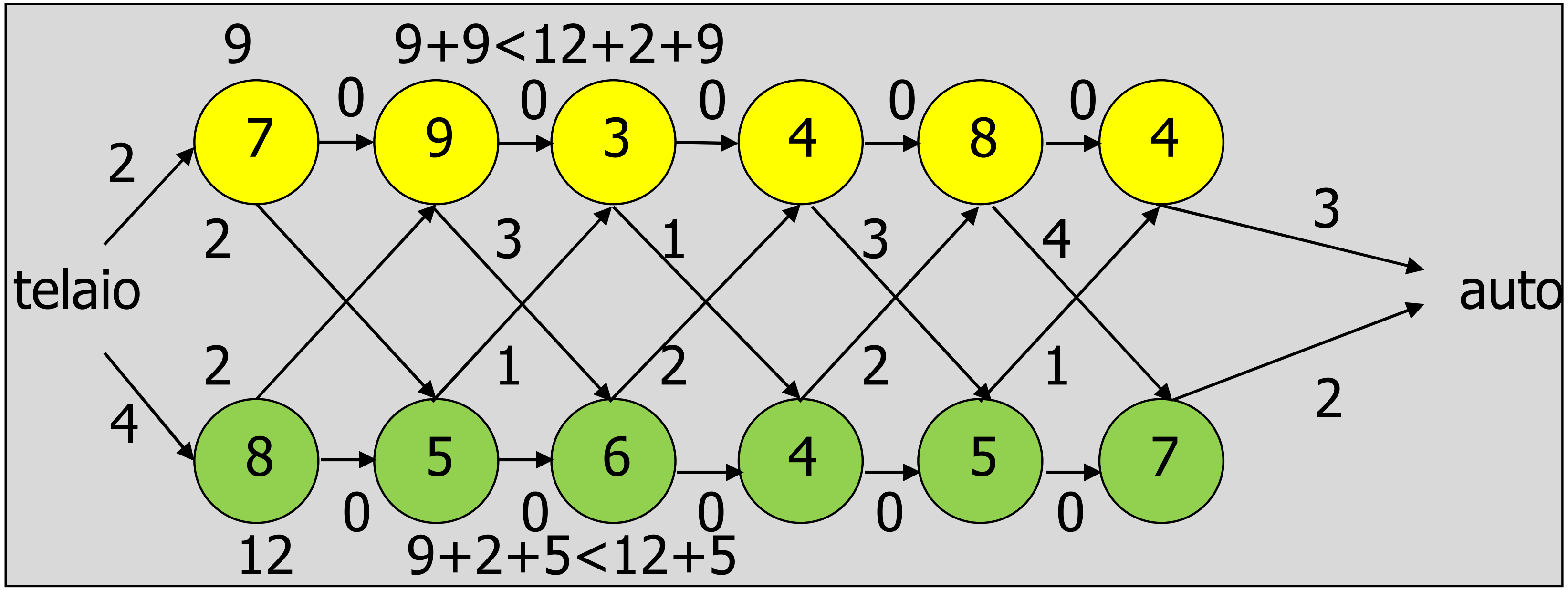


f

	0	1	2	3	4	5	6
0	9						
1	12						

l

	0	1	2	3	4	5	6
0							
1							

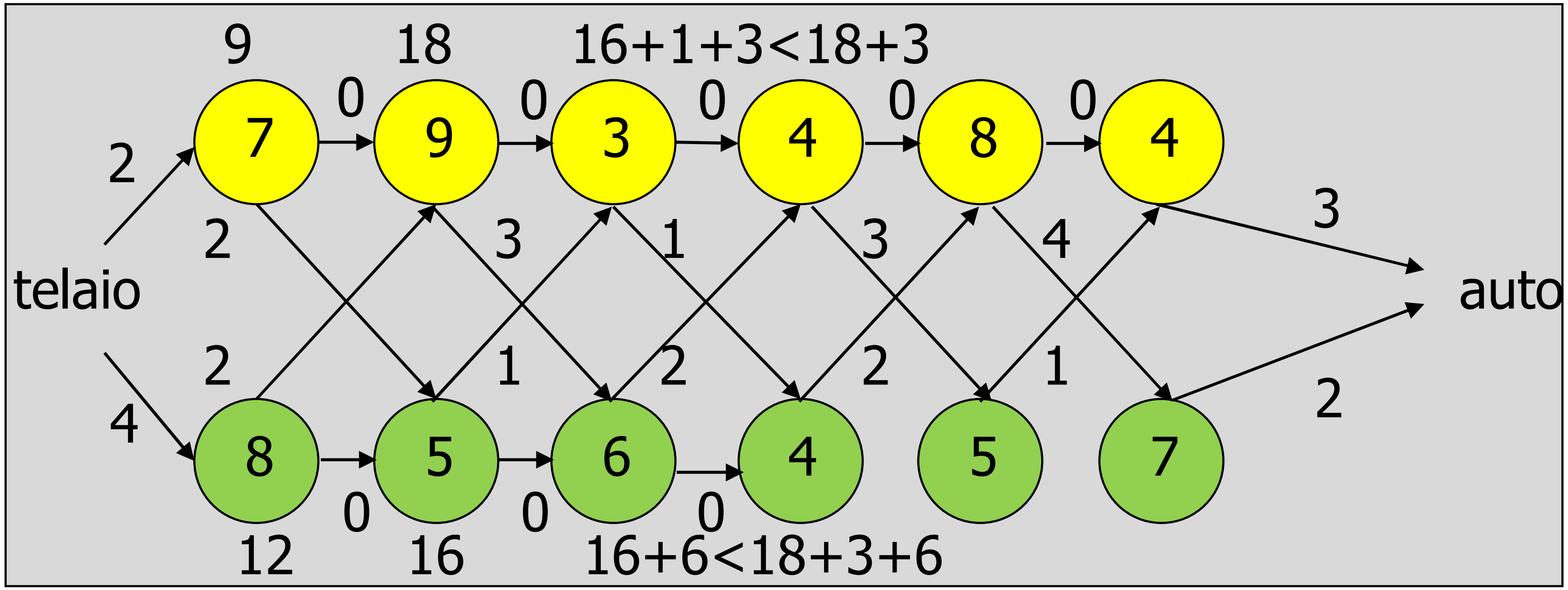


f

	0	1	2	3	4	5	6
0	9	18					
1	12	16					

l

	0	1	2	3	4	5	6
0		0					
1		0					

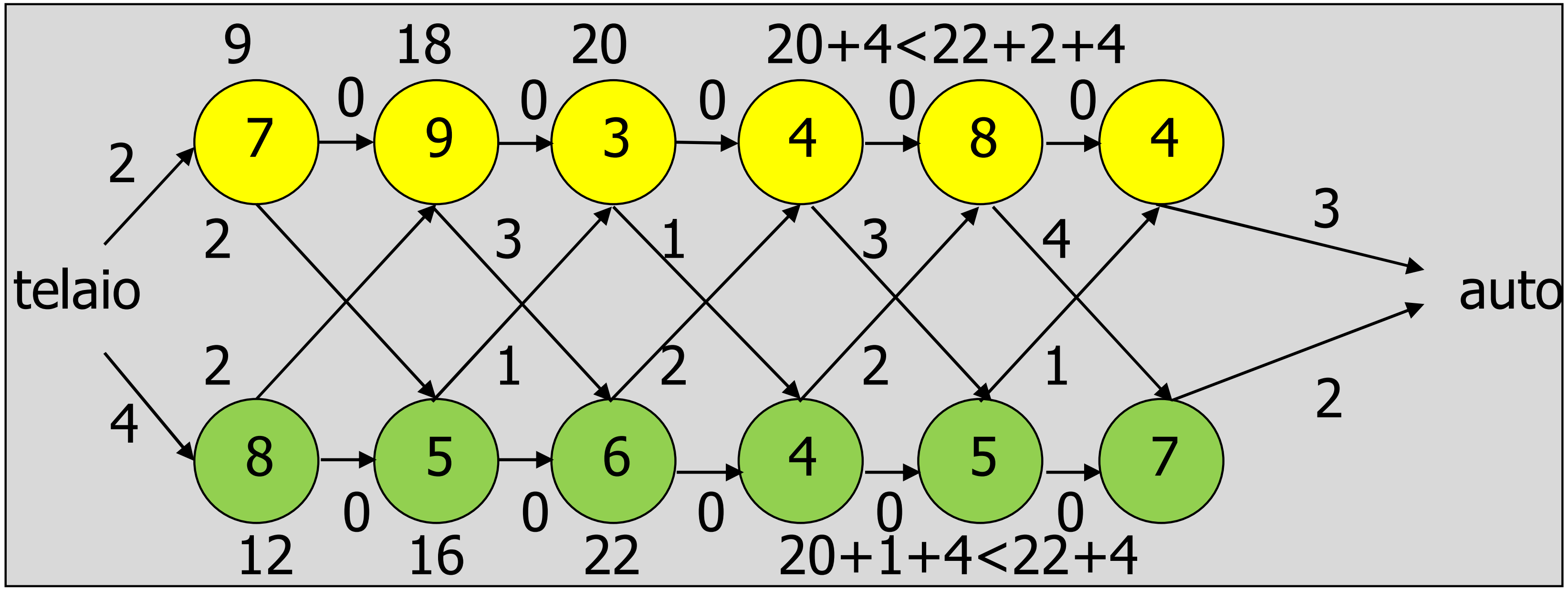


f

	0	1	2	3	4	5	6
0	9	18	20				
1	12	16	22				

l

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

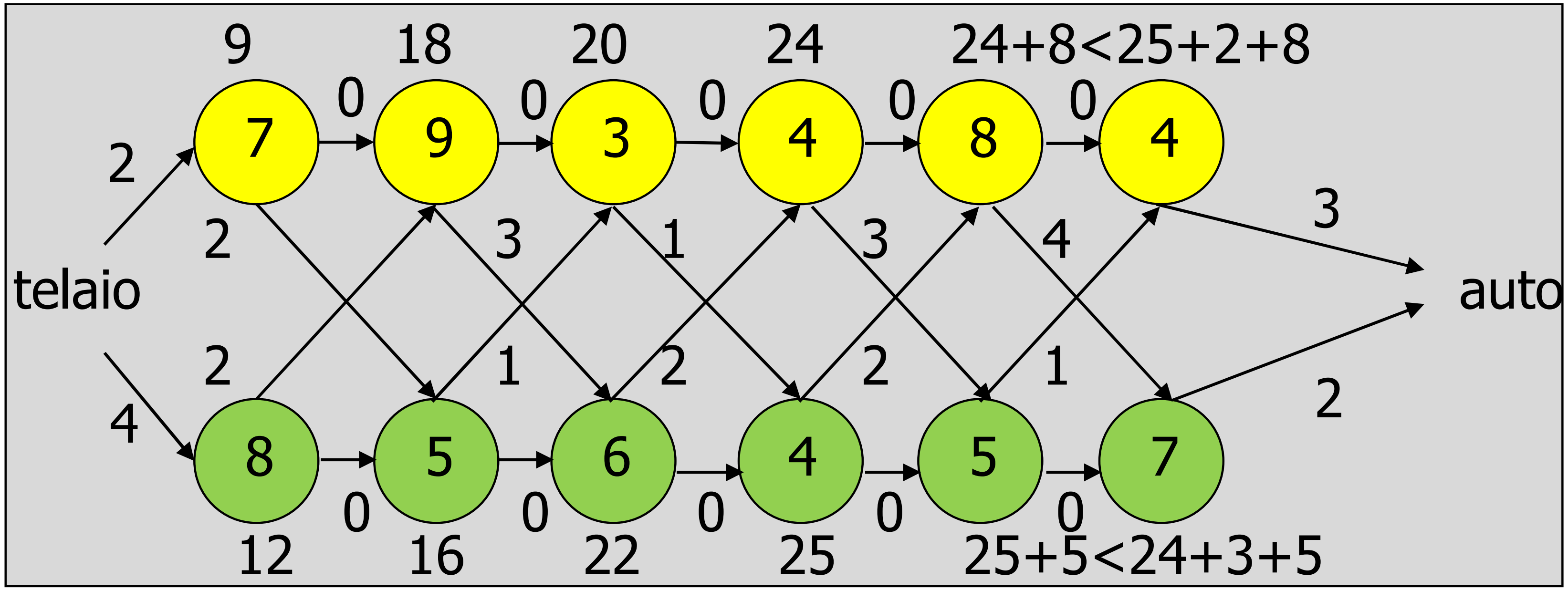


f

	0	1	2	3	4	5	6
0	9	18	20	24			
1	12	16	22	25			

l

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

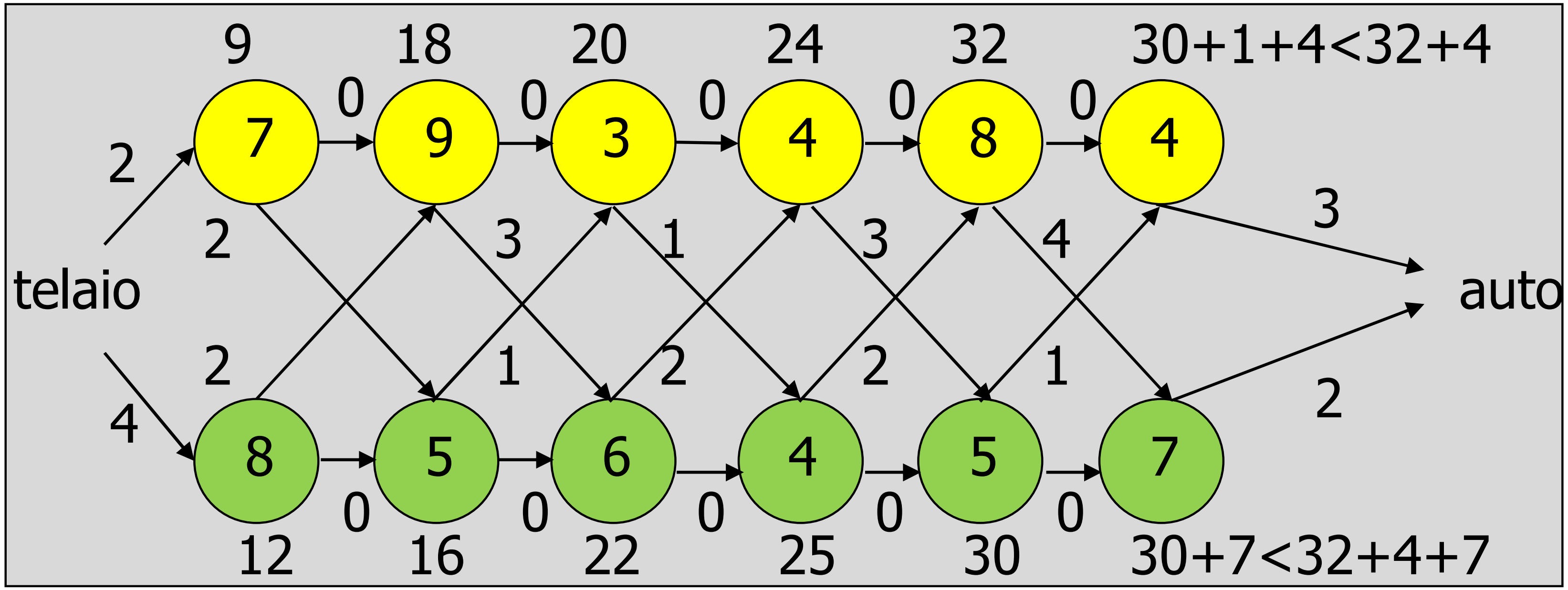


f

	0	1	2	3	4	5	6
0	9	18	20	24	32		
1	12	16	22	25	30		

l

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

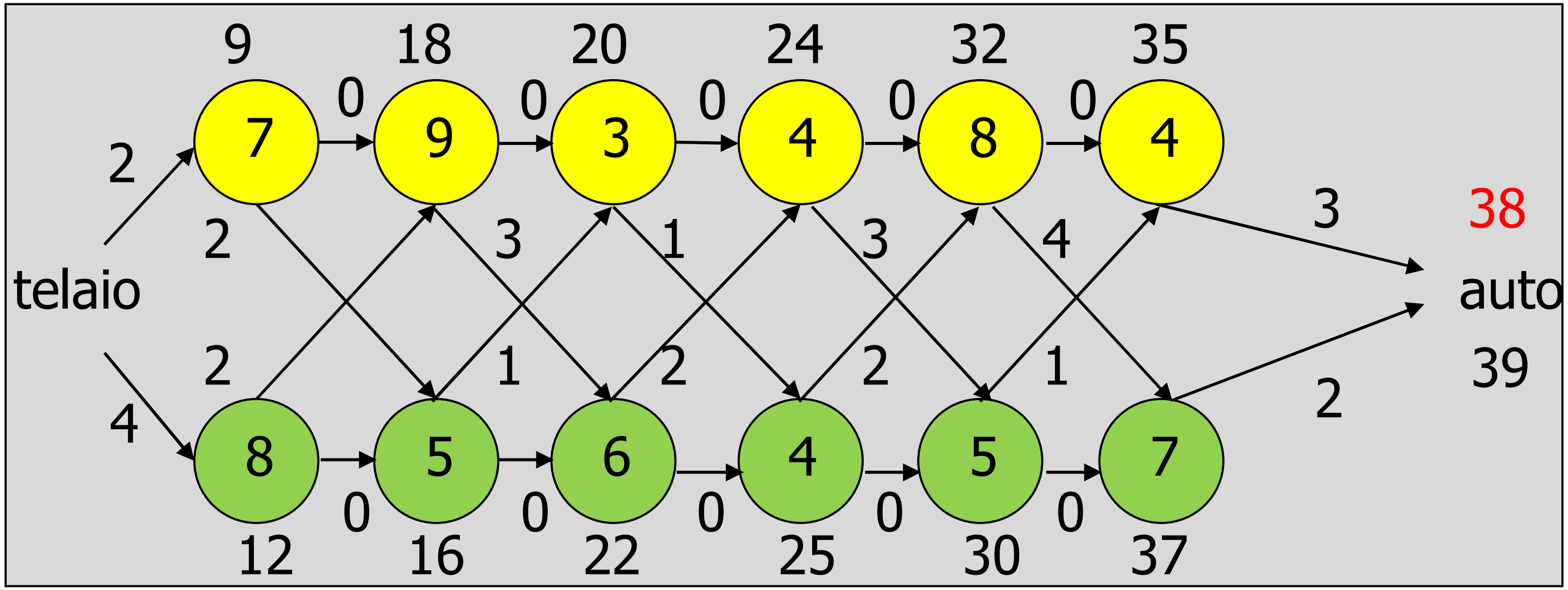


f

	0	1	2	3	4	5	6
0	9	18	20	24	32	35	
1	12	16	22	25	30	37	

l

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



f

	0	1	2	3	4	5	6
0	9	18	20	24	32	35	38
1	12	16	22	25	30	37	39

l

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

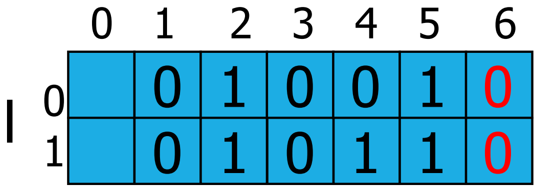
Complessità

$T(n) = \Theta(n)$
rispetto al costo esponenziale nel tempo della soluzione ricorsiva.

Per la Programmazione Dinamica in generale:
T(n) = Θ(numero di sottoproblemi x costo di ricombinazione delle loro soluzioni ottime).
In questo caso: n sottoproblemi, costo di ricombinazione unitario.

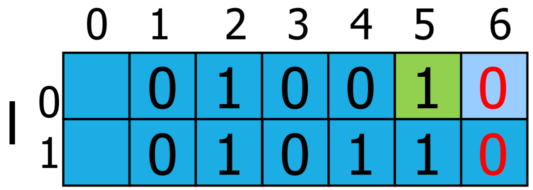
Costruzione di una soluzione ottima

```
void displaySol(int **l, int i, int n) {
  if (n==0)
    return;
  displaySol(l, l[i][n-1], n-1);
  printf("line %d station %d\n", i, n-1);
}
```



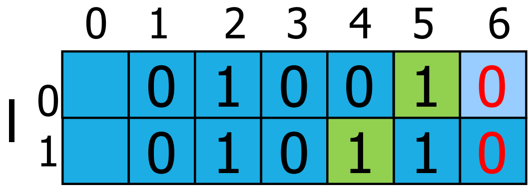
I	0	1	2	3	4	5	6
0		0	1	0	0	1	0
1		0	1	0	1	1	0

chiamata di `displaySol` con i=0



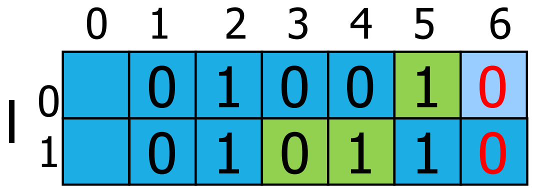
I	0	1	2	3	4	5	6
0		0	1	0	0	1	0
1		0	1	0	1	1	0

Soluzione = $\{S_{0,5}\}$



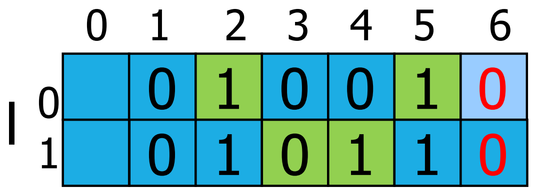
I	0	1	2	3	4	5	6
0		0	1	0	0	1	0
1		0	1	0	1	1	0

Soluzione = $\{S_{1,4}, S_{0,5}\}$



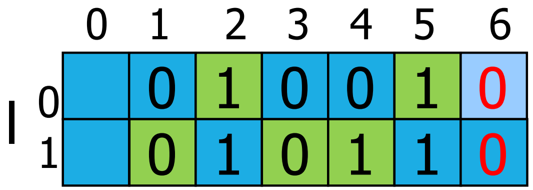
I	0	1	2	3	4	5	6
0		0	1	0	0	1	0
1		0	1	0	1	1	0

Soluzione = $\{S_{1,3}, S_{1,4}, S_{0,5}\}$



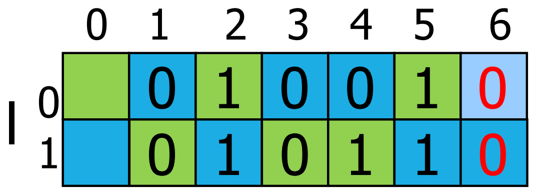
		0	1	2	3	4	5	6
I	0		0	1	0	0	1	0
	1		0	1	0	1	1	0

Soluzione = $\{S_{0,2}, S_{1,3}, S_{1,4}, S_{0,5}\}$



		0	1	2	3	4	5	6
I	0		0	1	0	0	1	0
	1		0	1	0	1	1	0

Soluzione = $\{S_{1,1}, S_{0,2}, S_{1,3}, S_{1,4}, S_{0,5}\}$



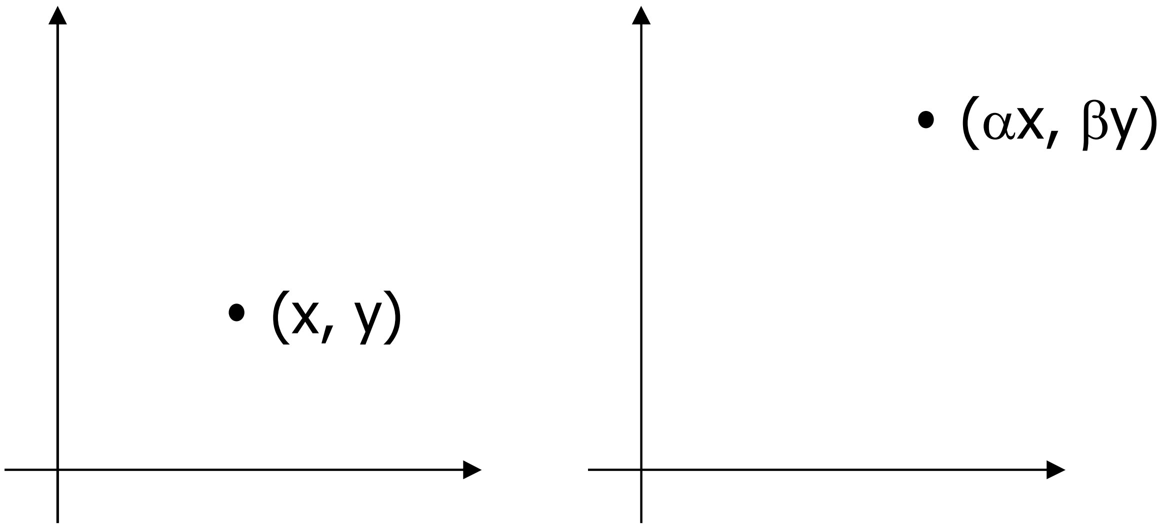
		0	1	2	3	4	5	6
I	0		0	1	0	0	1	0
	1		0	1	0	1	1	0

Soluzione = $\{S_{0,0}, S_{1,1}, S_{0,2}, S_{1,3}, S_{1,4}, S_{0,5}\}$

Grafica e moltiplicazione di matrici

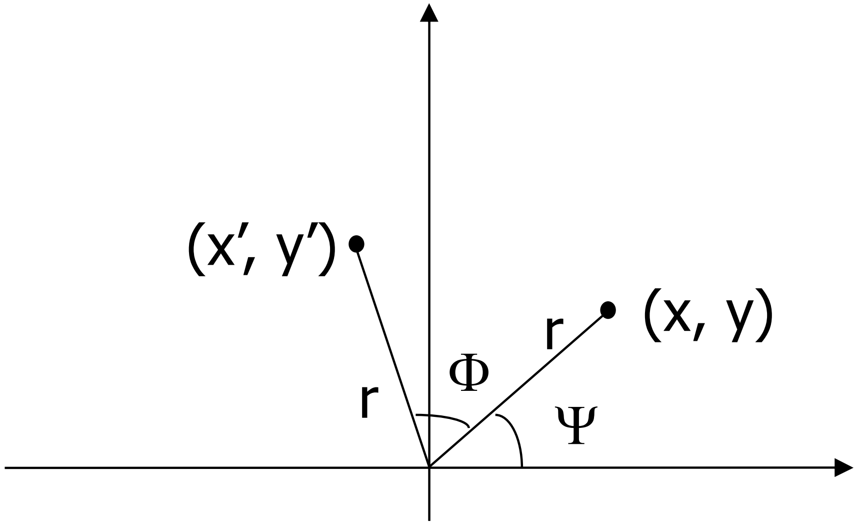
- Scena tridimensionale come insieme di triangoli nello spazio
- Triangolo individuato da 4 coordinate: assi x, y e z e dimensione fittizia (per scalamento)
- Operazioni grafiche elementari: scalamento, rotazione e traslazione di figure geometriche

Scalamento



$$[x, y, 1] \cdot \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{bmatrix} = [\alpha x, \beta y, 1]$$

Rotazione



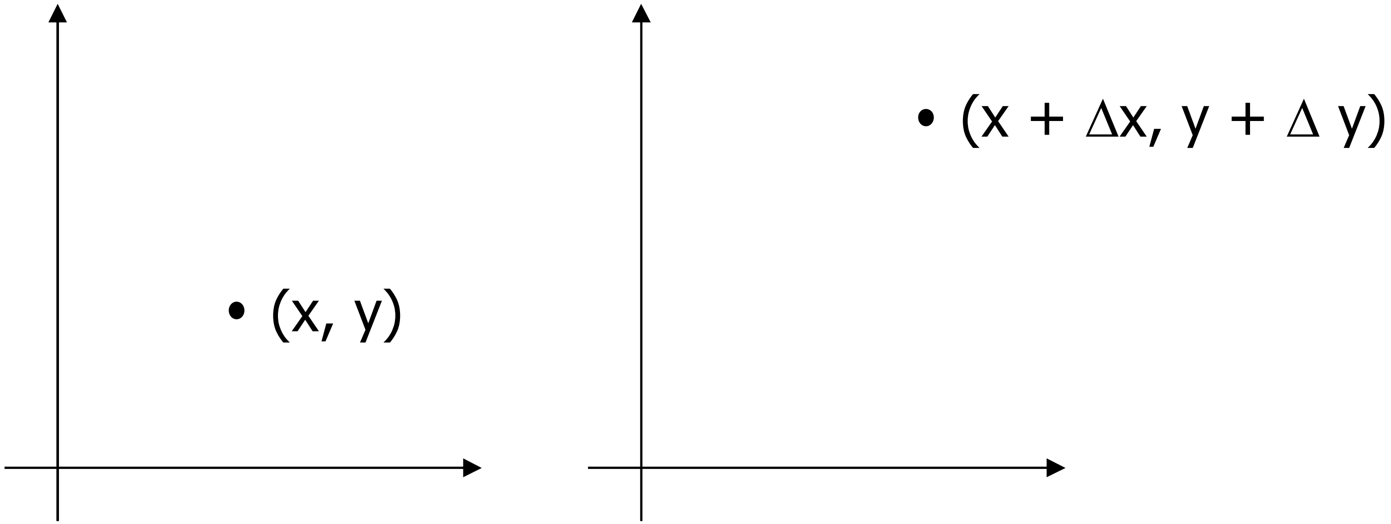
$x = r \cos \Psi$ $\qquad$ $\cos(\Phi+\Psi) = \cos\Phi\cos\Psi - \sin\Phi\sin\Psi$

$y = r \sin \Psi$ $\qquad$ $\sin(\Phi+\Psi) = \sin\Phi\cos\Psi + \cos\Phi\sin\Psi$

$$
\begin{aligned}
x' &= r\cos(\Phi+\Psi) \\
&= r\cos\Phi\cos\Psi - r\sin\Phi\sin\Psi \\
&= x\cos\Phi - y\sin\Phi \\
y' &= r\sin(\Phi+\Psi) \\
&= r\sin\Phi\cos\Psi + r\cos\Phi\sin\Psi \\
&= x\sin\Phi + y\cos\Phi
\end{aligned}
$$

$$
[x, y, 1] \cdot \begin{bmatrix} \cos\Phi & \sin\Phi & 0 \\ -\sin\Phi & \cos\Phi & 0 \\ 0 & 0 & 1 \end{bmatrix} = [x', y', 1]
$$

Traslazione



$$[x, y, 1] \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \Delta x & \Delta y & 1 \end{bmatrix} = [x+\Delta x, y+\Delta\, y,1]$$

Trasformazione:

$[x, y, z, 1] \cdot A_1 \cdot A_2 \; \ldots.. \cdot A_n$

Stessa trasformazione applicata a punti diversi $\Rightarrow$ calcolo una volta per tutte del prodotto

$$A_1 \cdot A_2 \; \ldots.. \cdot A_n$$

Prodotto di 2 matrici

- Due matrici A nr_1 x nc_1 e B nr_2 x nc_2 sono compatibili se e solo se $nc_1 = nr_2$
- Ipotesi di semplificazione: matrici quadrate di dimensione n x n
- Algoritmo semplice: 3 cicli annidati, complessità per matrici quadrate $T(n) = \Theta(n^3)$

```
void matMult(int **A,int **B,int **C,int nr1,int nc1,int nc2){
  int i, j, k;
  for (i=0; i<nr1; i++)
    for (j=0; j<nc2; j++) {
      C[i][j] = 0;
      for (k=0; k<nc1; k++)
        C[i][j] =  C[i][j] + A[i][k]*B[k][j];
    }
  }
```

02matr_mult

Prodotto in catena di n matrici

Data la sequenza di n matrici compatibili a 2 a 2 A_1, A_2, A_3, …. A_n dove A_i ha dimensioni p_{i-1} x p_i con i= 1, 2, …,n calcolare il prodotto

$$A_1 \cdot A_2 \cdot A_3 \cdot \ldots. \cdot A_n$$

Le dimensioni delle matrici sono memorizzate in un vettore di n+1 interi p.

Parentesizzazione

- Definisce l'ordine di applicazione delle operazioni di prodotto di due matrici con costo minimo
- Esempio: $A_1 \cdot A_2 \cdot A_3 \cdot A_4$ 5 parentesizzazioni possibili

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

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

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

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

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

Costi

Date A_1 p_0 x p_1 e A_2 p_1 x p_2, il costo di $A_1 \cdot A_2$ è legato al numero di moltiplicazioni scalari $p_0 x p_1 x\ p_2$

Esempio: $A_1 \cdot A_2 \cdot A_3$ dove A_1 10x100, A_2 100x 5, A_3 5x50

- Parentesizzazione #1: $(A_1 \cdot A_2) \cdot A_3$
  - costo di $A_1 \cdot A_2$ 10x100x5 = 5000, risultato A_{12} 10x5
  - costo di $A_{12} \cdot A_3$ 10x5x50 = 2500
  - costo totale 7500
- Parentesizzazione #2: $A_1 \cdot (A_2 \cdot A_3)$
  - costo di $A_2 \cdot A_3$ 100x5x50 = 25000, risultato A_{23} 100x50
  - costo di $A_1 \cdot A_{23}$ 10x100x50 = 50000
  - costo totale 75000

Numero di parentesizzazioni

Per n $\geq$ 2 la catena si può spezzare in 2 al punto k, con $1 \leq k \leq n-1$.
Il numero di parentesizzazioni è il prodotto del numero di parentesizzazioni delle 2 catene (la prima lunga k, la seconda lunga n-k)

$$P(n) = \begin{cases} 1 & n = 1 \\ \Sigma_{1 \leq k \leq n-1} P(k) \cdot P(n-k) & n \geq 2 \end{cases}$$

Si dimostra che P(n)= C(n-1)
dove C(n) è detto numero di Catalan e vale

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

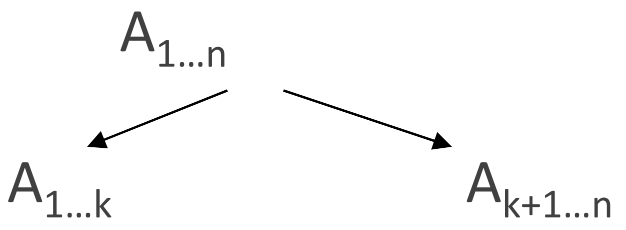
Struttura della soluzione ottima

è una matrice!

Notazione:

$$A_{i\ldots j} = A_i \cdot A_{i+1} \cdot \ldots\ldots A_j$$

Divisione in sottoproblemi



$A_{1\ldots n}$

$A_{1\ldots k}$ $A_{k+1\ldots n}$ $1 \leq k < n$

Costo di $A_{1\ldots n}$:

costo di $A_{1\ldots k}$ + costo di $A_{k+1\ldots n}$ + costo del prodotto $A_{1\ldots k} \cdot A_{k+1\ldots n}$

Perché sia ottima la soluzione di $A_{1\ldots n}$ devono essere ottime le soluzioni di $A_{1\ldots k}$ e $A_{k+1\ldots n}$.

Dimostrazione per assurdo: se la soluzione di $A_{1\ldots k}$ non fosse ottima, ne esisterebbe una migliore, quindi non sarebbe ottima la soluzione di $A_{1\ldots n}$. Analogamente per $A_{k+1\ldots n}$

Problema con sottostruttura ottima →
applicabilità del paradigma della programmazione dinamica

Soluzione ricorsiva

Valore della soluzione ottima:

sottoproblema: determinare il costo minimo della parentesizzazione di $A_{i...j}$ con $1 \leq i \leq j \leq n$.

m[i, j]: costo minimo per $A_{i...j}$

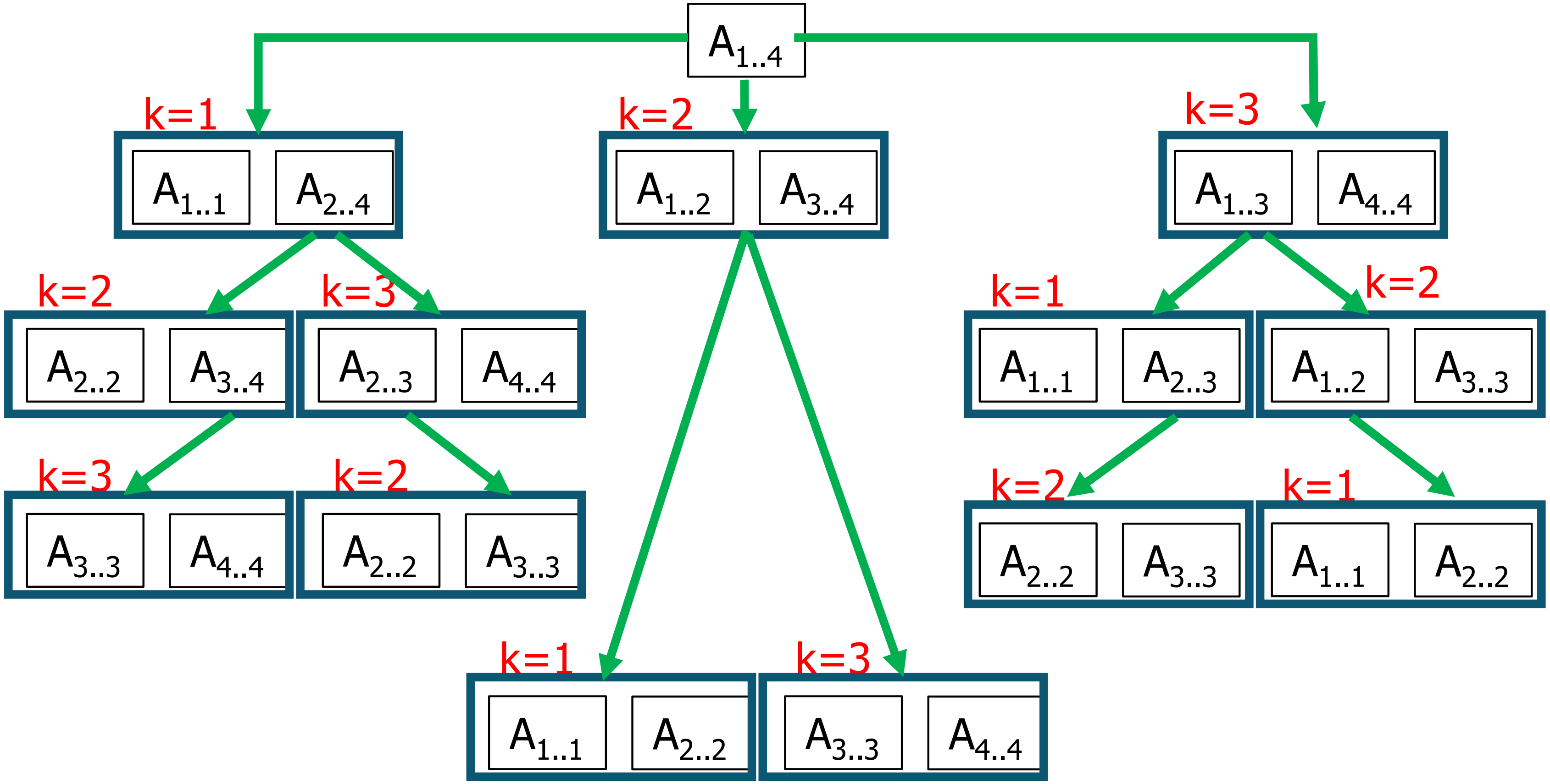
$$m[i, j] = \begin{cases} 0 & \text{se } i = j \\ \min\limits_{i \leq k < j}\{m[i,k]+m[k+1,j]+p_{i-1}p_k p_j\} & \text{se } i < j \end{cases}$$

s[i, j] contiene il valore di k che dà una parentesizzazione ottima nella divisione di $A_{i...j}$

solo calcolo del costo minimo

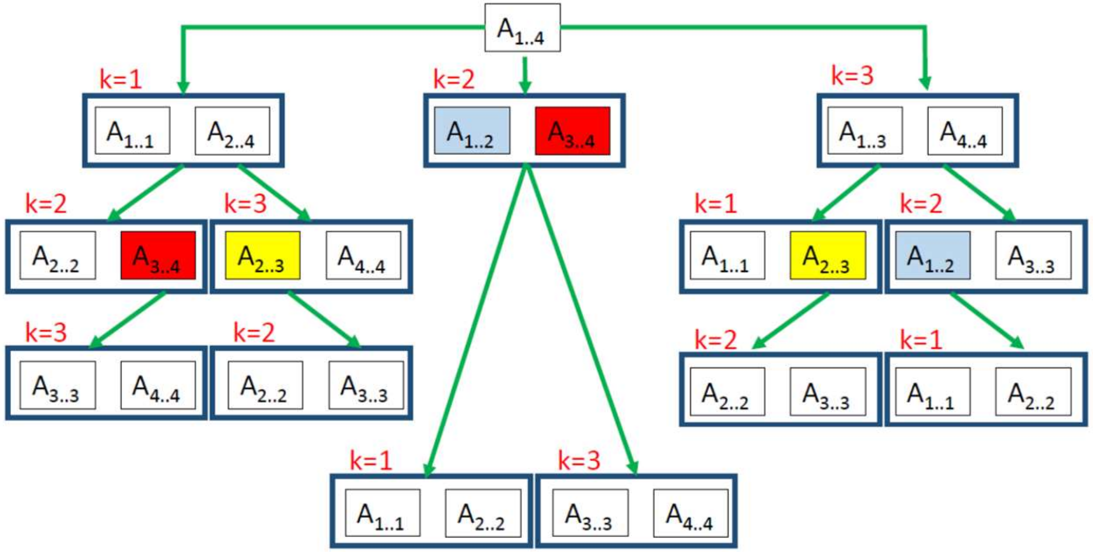
```c
int minCostR(int *p, int i, int j, int minCost) {
  int k, cost;
  if (i ==j)
    return 0;
  for (k=i; k<j; k++) {
    cost = minCostR(p, i, k, minCost) +
           minCostR(p, k+1, j, minCost) +
           p[i-1]*p[k]*p[j];
    if (cost < minCost)
      minCost = cost;
  }
  return minCost;
}

int matrix_chainR(int *p, int n) {
  return minCostR(p, 1, n, INT_MAX);
}
```

03matr_chain_mult


$A_{1..4}$
k=1
$A_{1..1}$ $A_{2..4}$
k=2
$A_{1..2}$ $A_{3..4}$
k=3
$A_{1..3}$ $A_{4..4}$
k=2
$A_{2..2}$ $A_{3..4}$
k=3
$A_{2..3}$ $A_{4..4}$
k=1
$A_{1..1}$ $A_{2..3}$
k=2
$A_{1..2}$ $A_{3..3}$
k=3
$A_{3..3}$ $A_{4..4}$
k=2
$A_{2..2}$ $A_{3..3}$
k=2
$A_{2..2}$ $A_{3..3}$
k=1
$A_{1..1}$ $A_{2..2}$
k=1
$A_{1..1}$ $A_{2..2}$
k=3
$A_{3..3}$ $A_{4..4}$

Limiti della soluzione ricorsiva

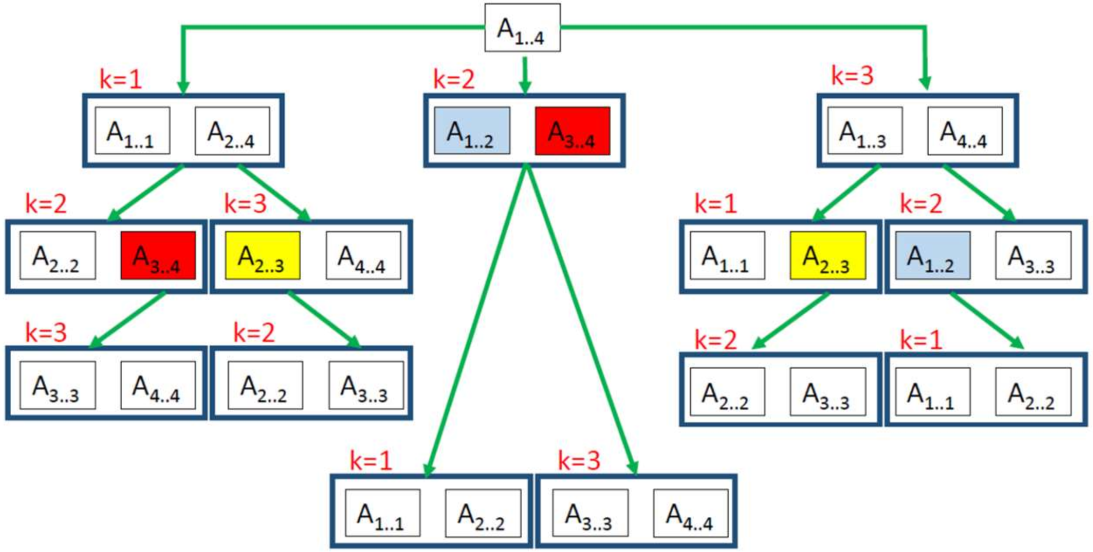
- assunzione di indipendenza dei sottoproblemi



- complessità: $T(n) = O(2^n)$

Numero di sottoproblemi indipendenti: combinazioni ripetute di n elementi presi a 2 a 2

$$C'_{n,2} = \frac{(n+2-1)!}{2!(n-1)!} = \frac{(n+1)!}{2!(n-1)!} = \frac{n(n+1)}{2} = \Theta(n^2)$$



Soluzione ottima: calcolo bottom-up del valore

Strutture dati:

- A_i matrice p_{i-1} x p_i con i=1, 2,….n
- vettore p delle dimensioni
- tabella m[1…n, 1…n] per memorizzare i costi m[i, j] e identificare il costo minimo
- tabella s[1…n, 1…n] per tenere traccia del valore ottimo di k per ricostruire la soluzione

non serve per il calcolo del costo minimo, servirà per costruire la soluzione

Passi:

- catene lunghe 1 ($A_{i \ldots j}$ con i=j): costi nulli m[i][j]=0 ∀ i=j
- catene lunghe 2 ($A_{1 \ldots 2}$, $A_{2 \ldots 3}$ …. $A_{n-1 \ldots n}$): nessuna scelta (k=i), costi fissi m[i][j]=p_{i-1} x p_i x p_j
- catene lunghe 3
  - $A_{1 \ldots 3}$: scelta per
    - k=1 usando m[1][1], m[2][3] e p_0 x p_1 x p_3
    - k=2 usando m[1][2], m[3][3] e p_0 x p_2 x p_3
  - $A_{2 \ldots 4}$: scelta per
    - k=2 usando m[2][2], m[3][4] e p_1 x p_2 x p_4
    - k=3 usando m[2][3], m[4][4] e p_1 x p_3 x p_4
  - $A_{n-2 \ldots n}$: scelta per
    - k=n-2 usando m[n-2][n-2], m[n-1][n] e p_{n-3} x p_{n-2} x p_n
    - k=n-1 usando m[n-2][n-1], m[n][n] e p_{n-3} x p_{n-1} x p_n
- etc. etc.

calcolo del costo minimo e della soluzione ottima

```
int matrix_chainDP(int *p, int n) {
  int i, l, j, k, q, **m, **s;

  m = calloc((n+1), sizeof(int *));
  s = calloc((n+1), sizeof(int *));

  for (i = 0; i <= n; i++) {
    m[i] = calloc((n+1), sizeof(int));
    s[i] = calloc((n+1), sizeof(int));
  }
```

costo 0 in quanto
catene lunghe 1

```
for (l = 2; l <= n; l++)
  for (i = 1; i <= n-l+1; i++) {
   j = i+l-1;
   m[i][j] = INT_MAX;
   for (k = i; k <= j-1; k++) {
    q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
    if (q < m[i][j]) {
     m[i][j] = q;
     s[i][j] = k;
    }
   }
  }
  displaySol(s, 1, n);
 return m[1][n];
}
```

ciclo su lunghezza crescente delle catene

identificazione di i e j

identificazione di k

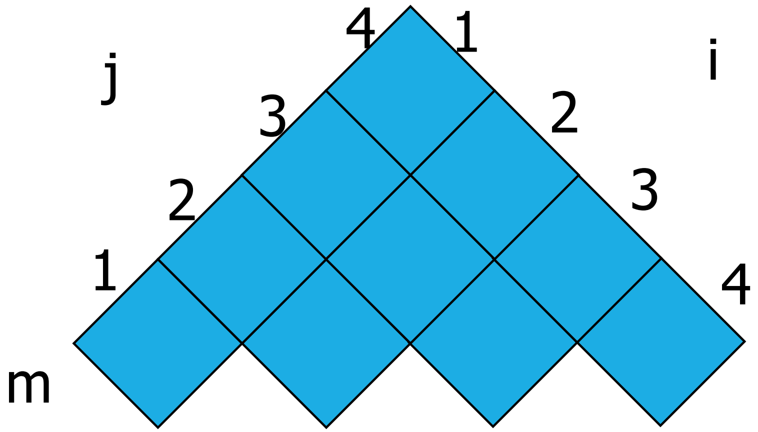
calcolo costo

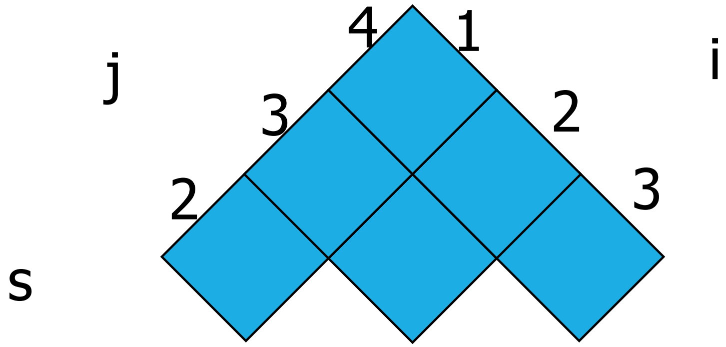
scelta

visualizzazione della soluzione ottima

Esempio

A_1	4 x 4
A_2	4 x 6
A_3	6 x 15
A_4	15 x 10





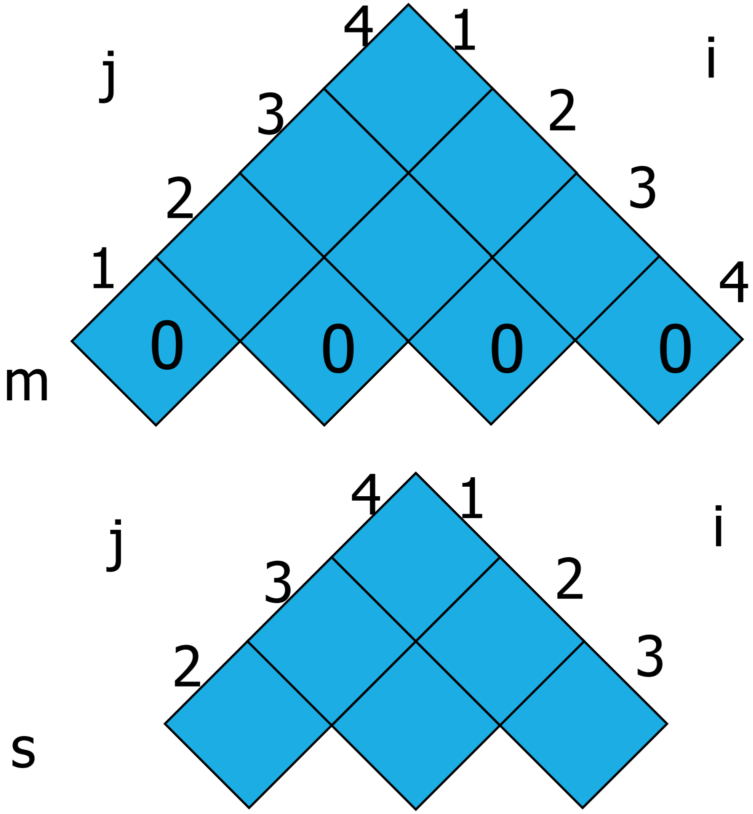
p	4	4	6	15	10

m[1,1] = 0

m[2,2] = 0

m[3,3] = 0

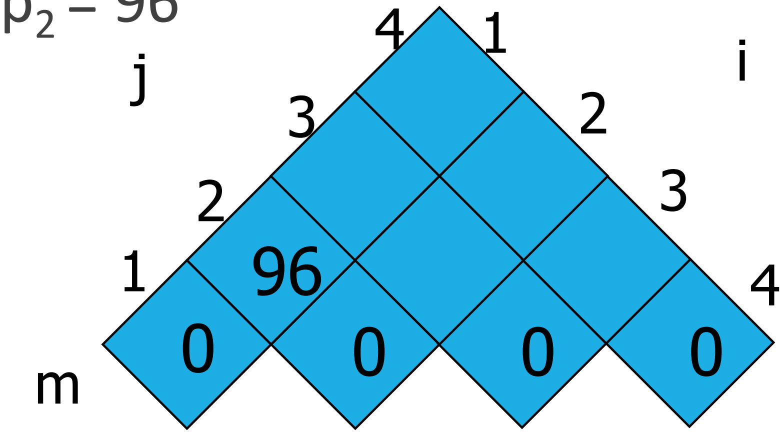
m[4,4] = 0

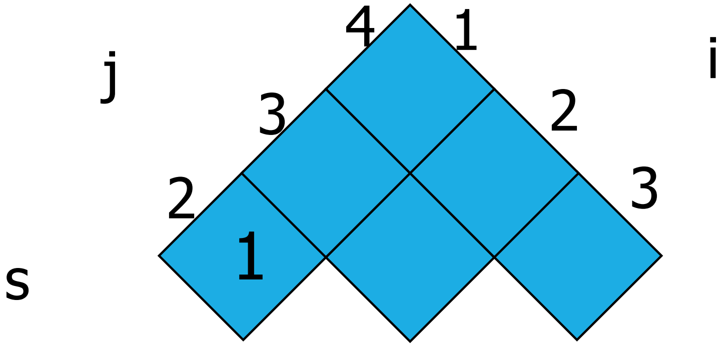


Catene lunghe 1

$m[1,2] = m[1,1] + m[2,2] + p_0p_1p_2 = 96$

k=1

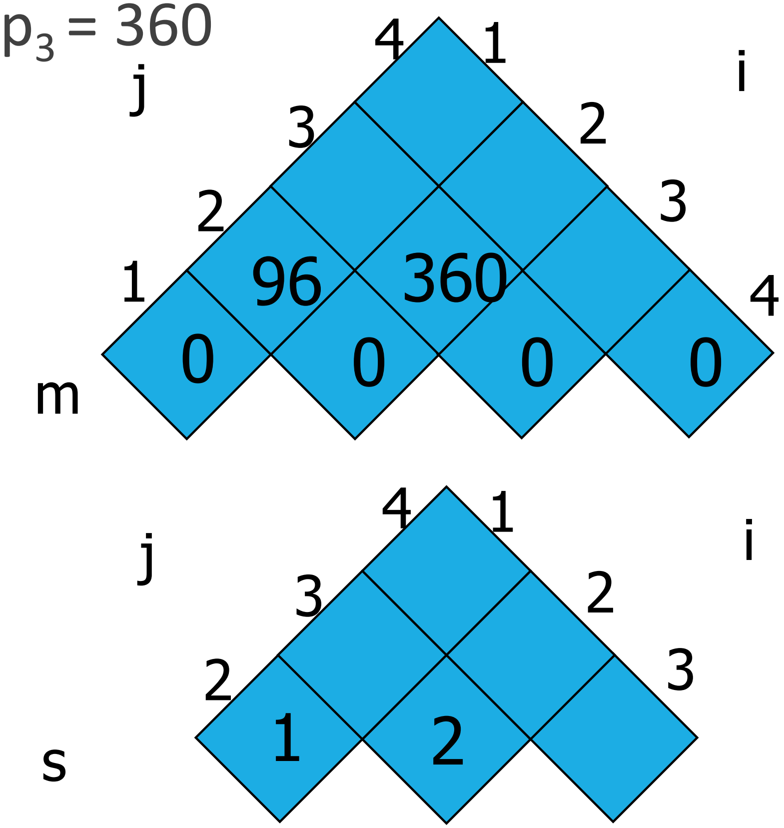




Catene lunghe 2: $A_{1..2}$

$m[2,3] = m[2,2] + m[3,3] + p_1p_2p_3 = 360$

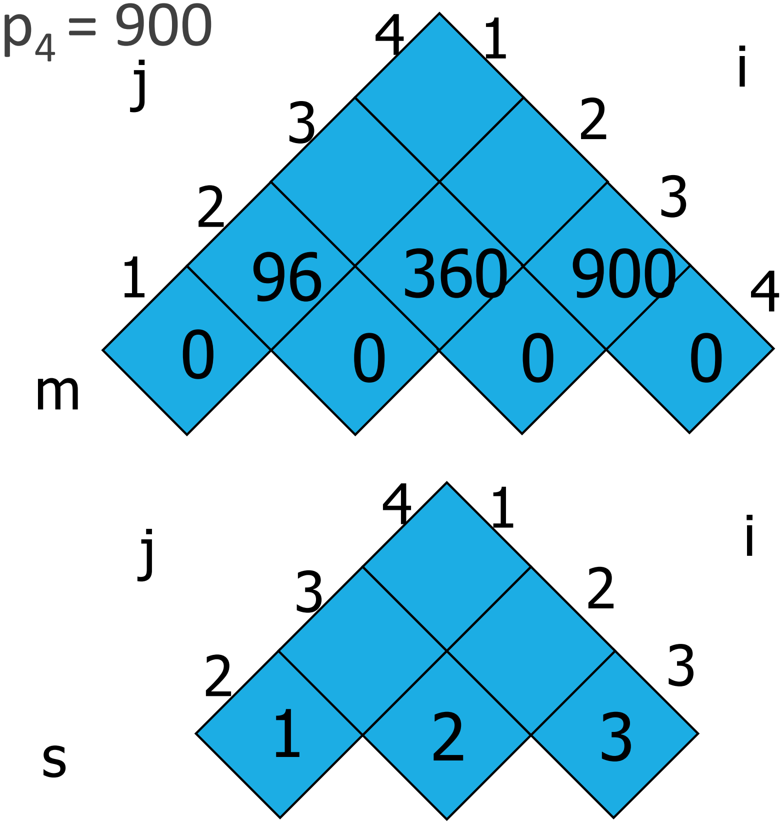
k=2



Catene lunghe 2: $A_{2..3}$

$m[3,4] = m[3,3] + m[4,4] + p_2 p_3 p_4 = 900$

$k=3$



Catene lunghe 2: $A_{3..4}$

$m[1,3] = m[1,1] + m[2,3]$

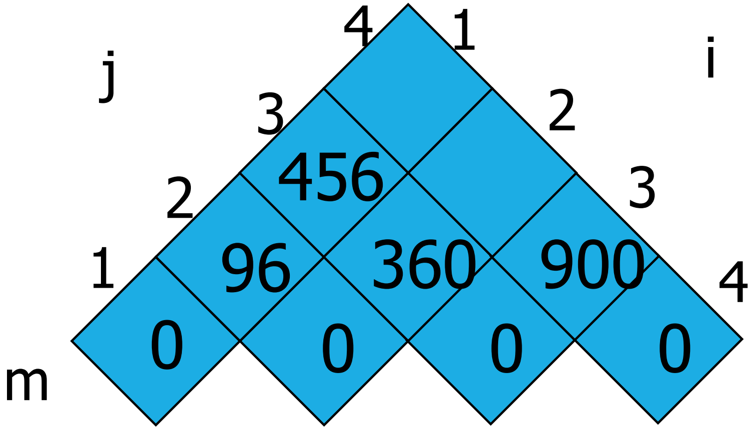
$+ p_0p_1p_3 = 360 + 240 = 600$

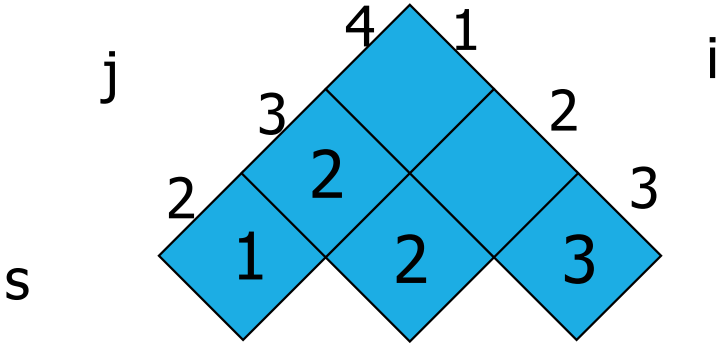
k=1

$m[1,3] = m[1,2] + m[3,3]$

$+ p_0p_2p_3 = 96 + 360 = 456$

k=2





Catene lunghe 3: $A_{1..3}$

$m[2,4] = m[2,2] + m[3,4]$

$+ p_1p_2p_4 = 900 + 240 = 1140$

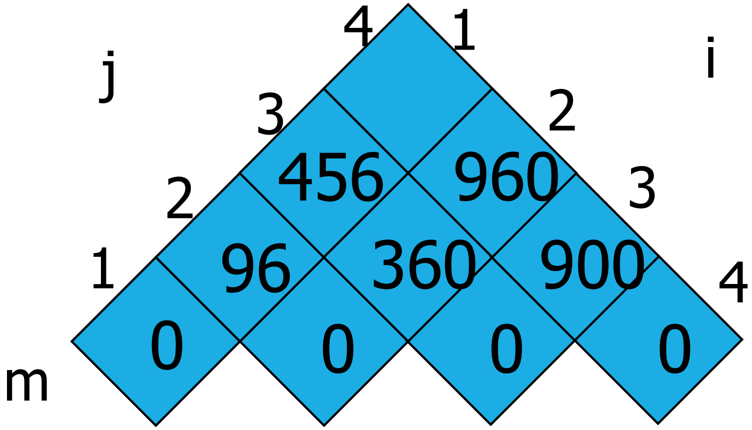
$k=2$

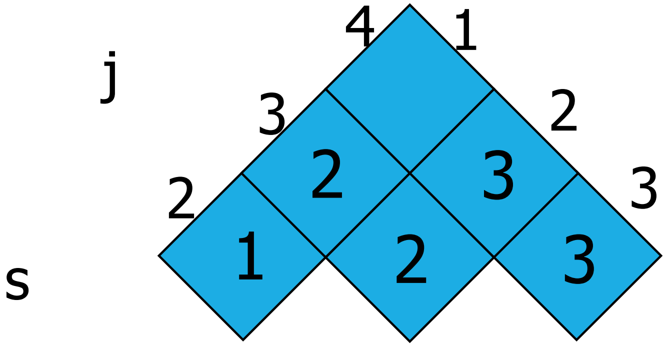
$m[2,4] = m[2,3] + m[4,4]$

$+ p_1p_3p_4 = 360 + 600 = 960$

$k=3$

Catene lunghe 3: $A_{2..4}$





Catena lunga 4: $A_{1..4}$

$m[1,4] = m[1,1] + m[2,4]$

$+ p_0p_1p_4 = 960 + 160 = 1120$

$k=1$

$m[1,4] = m[1,2] + m[3,4]$

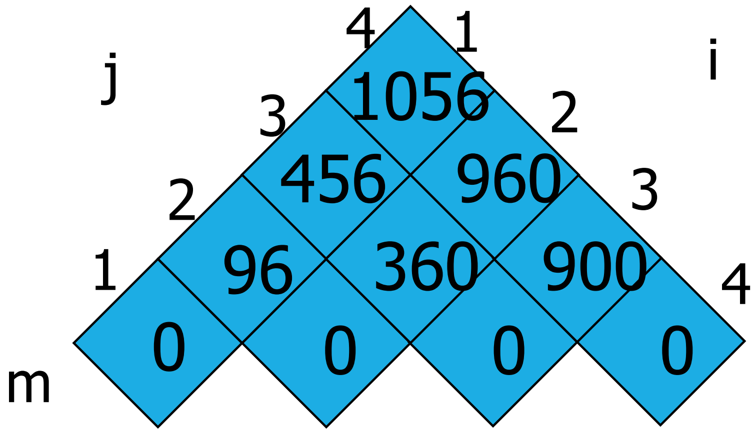
$+ p_0p_2p_4 = 1236$

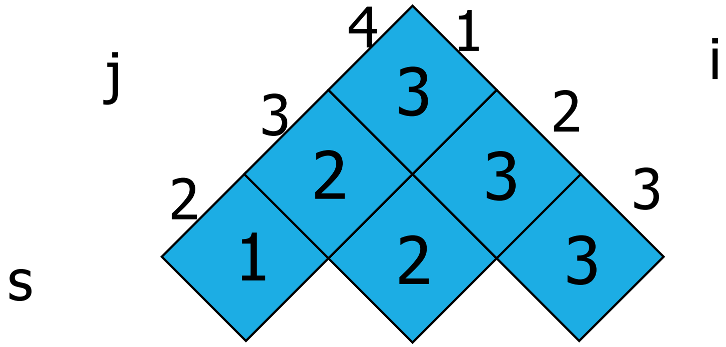
$k=2$

$m[1,4] = m[1,3] + m[4,4]$

$+ p_0p_3p_4 = 456 + 600 = 1056$

$k=3$





Complessità

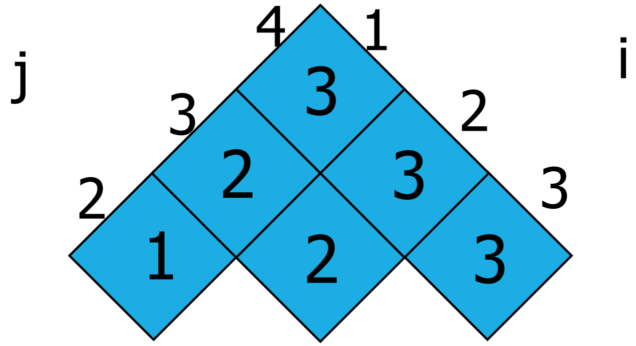
$T(n) = \Theta(n^3)$

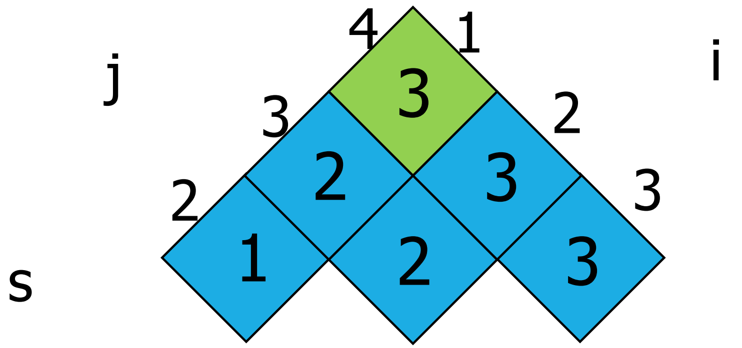
$S(n) = \Theta(n^2)$

rispetto al costo esponenziale nel tempo della soluzione ricorsiva

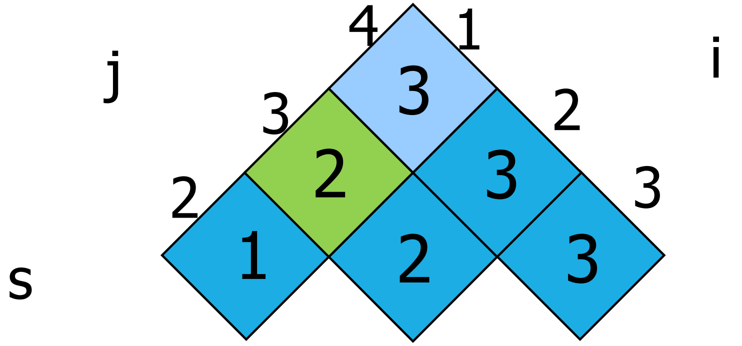
Soluzione ottima: costruzione

```
void displaySol(int **s, int i, int j) {
  if (j <= i) {
    printf("A%d", i);
    return;
  }
  printf("(");
  displaySol(s, i, s[i][j]);
  printf(") x (");
  displaySol(s, s[i][j]+1, j);
  printf(")");
  return;
}
```

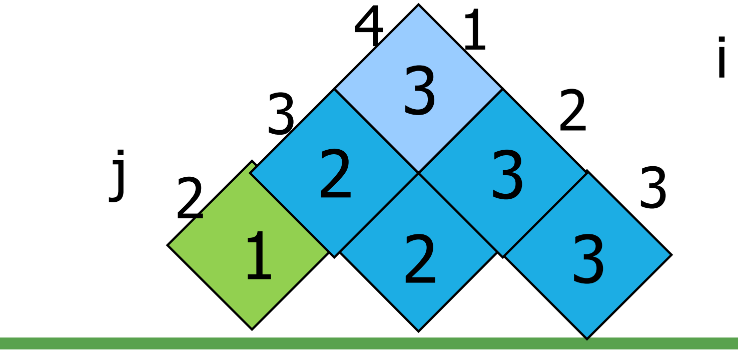




$$A_{1\ldots4} = (A_{1\ldots3}) \times A_4$$



$$\begin{aligned} A_{1\ldots4} &= (A_{1\ldots3}) \times A_4 \\ &= ((A_{1\ldots2}) \times A_3) \times A_4 \end{aligned}$$



$$\begin{aligned} A_{1\ldots4} &= (A_{1\ldots3}) \times A_4 \\ &= ((A_{1\ldots2}) \times A_3) \times A_4 \\ &= ((A_1)\times(A_2)) \times A_3) \times A_4 \end{aligned}$$

Applicabilità della programmazione dinamica

- Esistenza di una sottostruttura ottima
- Esistenza di molti sottoproblemi in comune:
  - vantaggio rispetto al divide et impera che assume l’indipendenza dei sottoproblemi
  - numero di sottoproblemi polinomiale nella dimensione dei dati in ingresso
  - sottoproblemi di complessità polinomiale
- Approccio bottom-up (parte da tutti i problemi elementari)

Esistenza della sottostruttura ottima

1. Dimostrare che una soluzione del problema consiste nel fare una scelta. Questa scelta genera uno o più sottoproblemi da risolvere
2. Per un dato problema, supporre di conoscere la scelta che porta a una soluzione ottima. Non interesse sapere come è stata determinata tale scelta
3. Fatta la scelta, determinare quali sottoproblemi ne derivano e quale sia il modo migliore per caratterizzare lo spazio di sottoproblemi risultante

4. Dimostrare per assurdo che le soluzioni dei sottoproblemi utilizzate all’interno della soluzione ottima del problema devono essere necessariamente ottime con la tecnica del «taglia & incolla»:
    - supporre che le soluzioni dei sottoproblemi non siano ottime
    - «tagliare» la sottosoluzione non ottima e «incollare» una sottosoluzione ottima
    - verificare che si è generata una contraddizione

Attenzione a non assumere l’esistenza della sottostruttura ottima se non è possibile farlo!

Esempio:
Dato un grafo orientato e non pesato e 2 suoi vertici u e v, trovare il cammino:

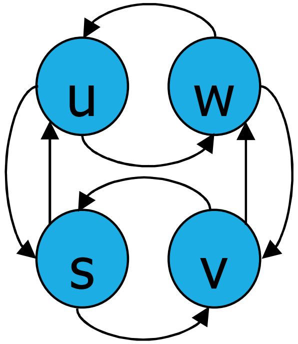
- minimo (formato dal minimo numero di archi). È necessariamente semplice: se non lo fosse, si potrebbe eliminare il ciclo e ridurre il numero di archi
- massimo: cammino semplice formato dal massimo numero di archi. Se non fosse semplice, percorrendo il ciclo infinite volte il problema perderebbe di significato

Cammino minimo tra u e v nel grafo G=(V, E):

- problema banale: u e v coincidono
- esiste un cammino minimo p: $u \rightarrow_p v$
    - essendo $u \neq v$, esiste un nodo intermedio w (può anche essere u o v) che spezza il cammino p in p_1 $u \rightarrow_{p1} w$ e p_2 $w \rightarrow_{p2} v$
    - se p_1 non fosse ottimo, esisterebbe p'_1 ottimo, che sostituito a p_1 porterebbe a concludere che p non è ottimo: contraddizione
    - analogamente per p_2

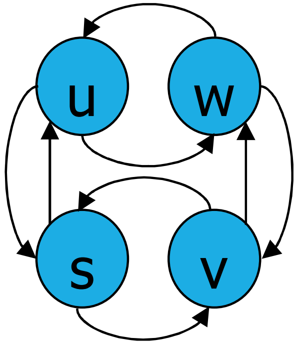
Cammino semplice massimo tra u e v nel grafo G=(V, E):

- (u, w, v) è un cammino massimo semplice tra u e v
- il suo sottocammino (u, w) non è massimo, è massimo il sottocammino (u, s, v, w)
- il suo sottocammino (w, v) non è massimo, è massimo il sottocammino (w, u, s, v)



Il problema non presenta una sottostruttura ottima!

Inoltre, combinando i sottocammini semplici e massimi (u, s, v, w) e (w, u, s, v) il cammino che si ottiene non è semplice



La programmazione dinamica non è applicabile.
Il problema dei cammini **massimi** è **NP-completo**.

Divide et impera vs. programmazione dinamica

Divide et impera

- tutti i tipi di problemi
- sottoproblemi indipendenti
- top-down
- ricorsione

Programmazione dinamica

- solo problemi di ottimizzazione
- sottoproblemi in numero limitato
- sottoproblemi condivisi
- bottom-up
- iterazione

Longest Increasing Sequence

Data una sequenza di N interi

$$X = (x_0, x_1, \dots x_{N-1})$$

si definisce **sottosequenza** di X di lunghezza k ($k \leq N$) un qualsiasi n-upla Y di k elementi di X con indici crescenti $i_0, i_1, \cdots, i_{k-1}$.

Si ricordi:

- sottosequenza: indici non necessariamente contigui
- sottostringa: indici contigui
- prefisso i-esimo di lunghezza i+1 di una sequenza X: $X_i = (x_0, x_1, \dots x_i)$

- Longest Increasing Sequence:
  - sottosequenza di lunghezza massimale
  - di elementi con valori crescenti
- Problema: determinare una LIS.
- Soluzioni:
  - ricorsione con modello del Calcolo Combinatorio (powerset) (T(n) = $O(2^N)$)
  - programmazione dinamica se
    - applicabile ⇒ sottostruttura ottima
    - conveniente ⇒ numero polinomiale di sottoproblemi indipendenti

Applicabilità/convenienza

- Esistenza di una sottostruttura ottima: data una soluzione ottima, se non fosse ottima la soluzione al problema per ogni prefisso (sottoproblema), se ne potrebbe trovare una migliore e di conseguenza la soluzione ottima non sarebbe tale (assurdo)
- Il numero di sottoproblemi indipendenti è polinomiale ($O(N)$) nella dimensione dei dati in ingresso
- La soluzione di ciascun sottoproblema è $O(N)$
- La soluzione con programmazione dinamica ha complessità $O(N^2)$

PS: esistono algoritmi $O(N\log N)$ che esulano da questo corso.

Soluzione ricorsiva

Valore della soluzione ottima:

- funzione ricorsiva LISR: dato il prefisso $X_i = (x_0, x_1, \dots x_i)$
  - se $i=0$, $c[X_i] = 1$
  - $\forall$ i $0 < i < N$ considero tutti i prefissi X_j con $0 \leq j < i$ che soddisfano la condizione $x_j < x_i$
    - l'elemento x_i può essere aggiunto in coda a formare una LIS più lunga di 1, calcolo $1 + c[X_j]$
    - prendo il massimo e lo assegno a $c[X_i]$

$$c[X_i] = \begin{cases} 1 & i=0 \\ \max(1 + c[X_j]) \text{ tale che } 0 \leq j < i \ \&\& \ x_j < x_i & i>0 \end{cases}$$

wrapper

```
int LIS(int *val) {
  return LISR(val, N-1);
}

int LISR(int *val, int i) {
  int j, ris;
  if (i == 0)
    return 1;
  ris = 1;
  for (j=0; j < i; j++)
    if (val[j] < val[i])
      ris = max(ris, 1 + LISR(val, j));
  return ris;
}
```

$c[X_0]=1$

$\max(1+c[X_j])$ tale che $0 \leq j<i$ && $x_j<x_i$

04LIS

Soluzione ottima: calcolo bottom-up del valore

Strutture dati:

- `val` vettore di input di N interi
- `L` vettore di N interi per memorizzare la lunghezza della LIS per ogni prefisso i-esimo
- `P` vettore di N interi per memorizzare l'indice dell'elemento precedente nella LIS
- `last` intero per memorizzare l'indice dell'ultimo elemento nella LIS

servono per costruire la soluzione

Passi:

- il valore minimo di una LIS è 1 (sequenza monotona decrescente)
- per lunghezze crescenti del sottovettore considerato (prefisso i da 1 a N-1)
  - individuare la LIS del prefisso i che soddisfa le condizioni
  - registrare in P l’indice del valore precedente
- stampa ricorsiva dei valori mediante percorrimento di P

Esempio

val	11	7	13	15	8	14
L	1					
P	-1					

i=0

$c[X_0] = 1$

val	11	7	13	15	8	14
L	1	1				
P	-1	-1				

i=1

$c[X_1] = 1$

val	11	7	13	15	8	14
L	1	1	2			
P	-1	-1	0			

i=2

$c[X_2] = 2$

val	11	7	13	15	8	14
L	1	1	2	3		
P	-1	-1	0	2		

i=3

$c[X_3] = 3$

val	11	7	13	15	8	14
L	1	1	2	3	2	
P	-1	-1	0	2	1	

i=4

$c[X_4] = 3$

val	11	7	13	15	8	14
L	1	1	2	3	2	3
P	-1	-1	0	2	1	2

i=5

$c[X_5] = 3$

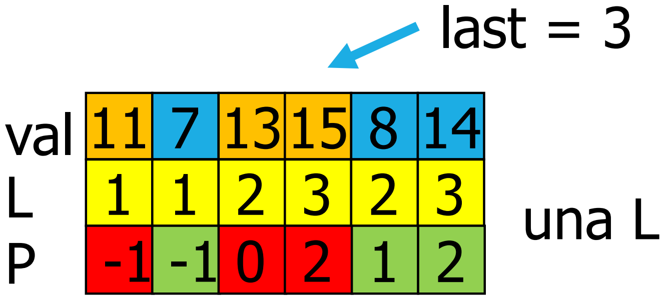
```c
void LISDP(int *val) {
  int i, j, ris=1, L[N], P[N], last=1;
  L[0] = 1; P[0] = -1;
  for (i=1; i<N; i++) {
    L[i] = 1; P[i] = -1;
    for (j=0; j<i; j++)
      if ((val[j] < val[i]) && (L[i] < 1 + L[j])) {
          L[i] = 1 + L[j]; P[i] = j;
        }
    if (ris < L[i]) {
      ris = L[i]; last = i;
    }
  }
  printf("One of the Longest Increasing Sequences is ");
  LISprint(val, P, last);
  printf("and its length is %d\n", ris);
}
```

calcolo del costo minimo e
di una soluzione ottima

visualizzazione di una
soluzione ottima

```c
#define N 7
```

```
void LISprint(int *val, int *P, int i) {
  if (P[i]==-1) {
    printf("%d ", val[i]);
    return;
  }
  LISprint(val, P, P[i]);
  printf("%d ", val[i]);
}
```



last = 3

val	11	7	13	15	8	14
L	1	1	2	3	2	3
P	-1	-1	0	2	1	2

una LIS = (11, 13, 15)

Longest Common Subsequence

Date 2 sequenze X e Y, Z è una sottosequenza comune se è sottosequenza sia di X che di Y.

Esempio:

X =

A	C	G	C	T	A	C
0	1	2	3	4	5	6

Y =

C	T	G	A	C	A
0	1	2	3	4	5

Sottosequenza comune:

C	G	A

Sottosequenze comuni di lunghezza massima (LCS):

C	G	C	A

C	T	A	C

La determinazione della LCS trova applicazione nei confronti di DNA.

Essa è trattata nei lucidi di approfondimento disponibili sul Portale della Didattica.

05LCS

Lo zaino (discreto)

Dato un insieme di N oggetti ciascuno dotato di peso w_j e di valore v_j e dato un peso massimo cap, determinare il sottoinsieme S di oggetti tali che:

- $\sum_{j \in S} w_j x_j \leq \text{cap}$
- $\sum_{j \in S} v_j x_j = \text{MAX}$
- $x_j \in \{0,1\}$

Ogni oggetto o è preso ($x_j = 1$) o lasciato ($x_j = 0$).

Ogni oggetto esiste in una sola istanza.

Esempio

3

Tipologia 3 nelle slide di programmazione

N= 4

cap = 10

item	name	size	value

Nome	A	B	C	D
Valore v_i	10	6	8	9
Peso w_i	8	4	2	3

Soluzione:

insieme {B, C, D} con valore massimo 23

Struttura della soluzione ottima

- Problema P(N, cap)
- Sottoproblema P(i, cap): problema per i primi i oggetti
- S(i, cap): soluzione ottima per P(i, cap)
- 2 casi:
  - l’oggetto i (ultimo degli oggetti correnti) appartiene alla soluzione ottima
  - l’oggetto i non appartiene alla soluzione ottima

Dimostrazione per assurdo:

I. l’oggetto i appartiene a S(i, cap) $\Rightarrow$ S(i,cap)-{i} è una soluzione ottima

   se non lo fosse, allora esisterebbe una soluzione S’ con valore maggiore e compatibile con la capacità. Se a S’ si riaggiungesse l’oggetto i, allora la soluzione risultante, certamente compatibile con la capacità, avrebbe valore maggiore della soluzione di partenza, contraddicendo l’ipotesi di soluzione ottima.

II. l’oggetto i non appartiene a S(i, cap) $\Rightarrow$ S(i,cap) è una soluzione ottima per P(i-1, cap)

   se non lo fosse, allora esisterebbe una soluzione S’ con valore maggiore e compatibile con la capacità, contraddicendo l’ipotesi di soluzione ottima.

Il numero di sottoproblemi indipendenti è $\Theta(N \cdot cap)$: la programmazione dinamica è conveniente.

Si identificano 2 fasi:

- ottimizzazione: calcolo del massimo valore compatibile con la capacità
- ricerca: identificazione degli elementi

Approccio ricorsivo "divide et impera" al solo problema di ottimizzazione.

Soluzione ricorsiva

Valore della soluzione ottima:

- caso terminale: non ci sono oggetti (i<0) o la capacità disponibile è nulla (j=0)
- caso non terminale:
  - l’oggetto i-esimo non può essere preso perché, aggiungendolo alla soluzione, si eccede la capacità
  - l’oggetto i-esimo può essere preso, ma non è detto che convenga
    - se conviene, l’oggetto viene preso
    - altrimenti l’oggetto non viene preso.

Valutazione delle convenienza: termini di paragone:

- maxv[i-1,j]: massimo valore con capacità disponibile j considerando gli oggetti da 0 a i-1, quindi non l’oggetto i
- v[i] + maxv[i-1,j-w[i]]: valore dell’oggetto i sommato al massimo valore ottenuto considerando gli oggetti da 0 a i-1 e una capacità disponibile j-w[i] tale da poter contenere l’oggetto i di peso w[i]

Se maxv[i-1,j] ≥ v[i] + maxv[i-1,j-w[i]]

- non prendo l’oggetto i e maxv[i,j] = maxv[i-1,j]
- altrimenti prendo l’oggetto i e maxv[i,j] = v[i] + maxv[i-1,j-w[i]].

massimo valore di i oggetti
con capacità disponibile j

non ci sono oggetti
o capacità disponibile 0

peso dell'oggetto
i eccede capacità
disponibile j

$$\text{maxv}[i, j] = \begin{cases} 0 & \text{se } i<0 \text{ o } j=0 \\ \text{maxv}[i-1,j] & \text{se } w[i] > j \\ \max\{\text{maxv}[i-1,j], \text{maxv}[i-1,j-w[i]]+v[i]) & \text{se } w[i] <= j \end{cases}$$

l'oggetto i non
viene preso

se è possibile prendere l'oggetto i, prenderlo
se migliora il valore rispetto a non prenderlo

```
int maxValR(Item *items, int i, int j) {
  if ( (i < 0) || (j == 0))
    return 0;
  if (items[i].size >j)
    return maxValR(items, i-1, j);
  return max(maxValR(items, i-1, j),
             maxValR(items,i-1,j-items[i].size)+items[i].value);
}

int KNAPmaxVal(Item *items, int N, int cap) {
  return maxValR(items, N, cap);
}
```

06knapsack

Soluzione ottima: calcolo bottom-up del valore

N oggetti identificati da indici 1,2,…, N. Oggetto fittizio all’indice 0 di peso e valore 0.
Capacità crescenti da 0 a cap. Zaino fittizio all’indice 0 con capacità 0.

Strutture dati:

- vettore items di N elementi
- tabella (N+1)x(cap+1) maxVal[0…N, 0…cap] per memorizzare i valori e identificare il massimo
- non serve un’ulteriore struttura dati per la costruzione dalla soluzione

Esempio

	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

N = 4
oggetti 1,2,3 4
cap = 10

j

i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0

oggetto fittizio

zaino fittizio

maxval

Passi:

- 2 cicli `for` annidati di scansione degli N oggetti (i da 1 a N) e della capacità (j da 1 a cap)
- l’oggetto corrente i si trova in items[i-1]
- se il peso dell’oggetto corrente items[i-1].size eccede la capacità disponibile j, l’oggetto non viene preso e maxval[i, j] = maxval[i-1, j]
- altrimenti si valuta se l’oggetto corrente conviene come nella soluzione ricorsiva.

Valutazione delle convenienza: termini di paragone:

- maxval[i-1,j]: massimo valore con capacità disponibile j considerando gli oggetti da 0 a i-1, quindi non l’oggetto i
- items[i-1].value + maxv[i-1,j-items[i-1].size]: valore dell’oggetto i sommato al massimo valore ottenuto considerando gli oggetti da 0 a i-1 e una capacità disponibile j-items[i-1].size tale da poter contenere l’oggetto i di peso items[i-1].size

Se maxval[i-1,j] ≥ maxval[i-1][j-items[i-1].size] + items[i-1].value

- non prendo l’oggetto i e maxval[i,j] = maxval[i-1,j]
- altrimenti prendo l’oggetto i e
  maxval[i,j] = maxval[i-1][j-items[i-1].size] + items[i-1].value

```
int KNAPmaxValDP(Item *items, int N, int cap) {
  int i, j, **maxval;
  // allocazione matrice maxval di dimensioni (N+1)x(cap+1)
  for (i=1; i<=N; i++)
    for (j=1; j <=cap; j++) {
      if (items[i-1].size > j)
        maxval[i][j] = maxval[i-1][j];
      else
        maxval[i][j] = max(maxval[i-1][j],
                           maxval[i-1][j-items[i-1].size] +
                           items[i-1].value);
    }
}
  printf("Maximum booty is: \n");
  displaySol(items, N, cap, maxval);
  printf("\n");
  return maxval[N][cap];
}
```

calcolo del valore massimo e di una soluzione ottima

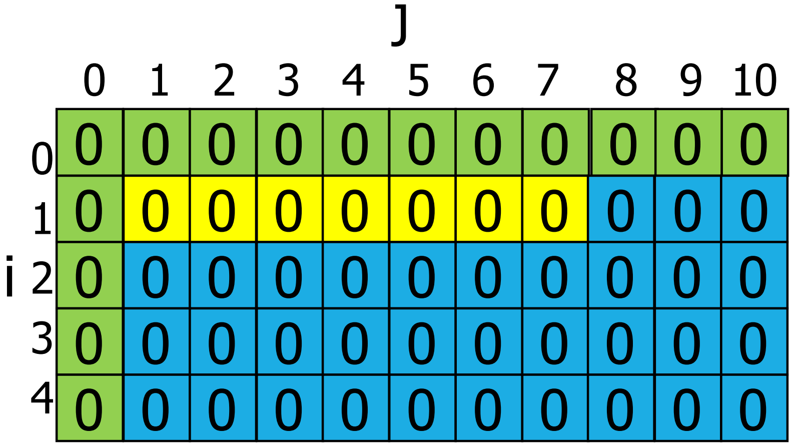
visualizzazione di una soluzione ottima

Esempio

Verde: celle con valore già calcolato 🟩
Giallo: celle con valore da calcolare 🟨
Rosso: celle di confronto 🟥

	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

Oggetto 1 🟥 (i=1) valore 10, peso 8



i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0

maxval

$\forall j \; 1 \leq j \leq 7$

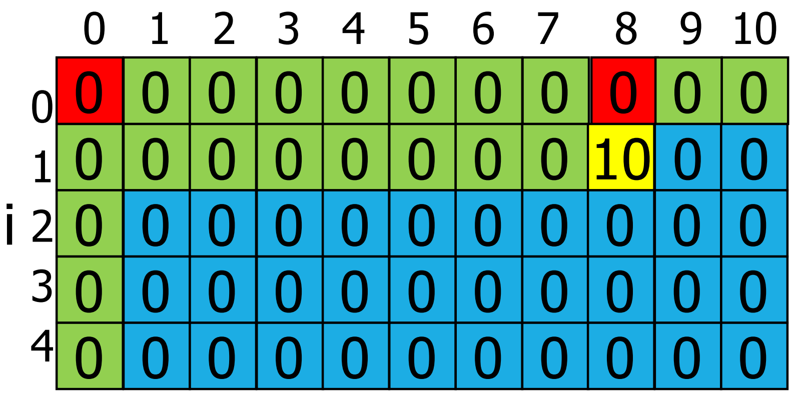
items[i-1].size > j $\Rightarrow$ l’oggetto 1 non viene preso

$\Rightarrow$ maxval[i, j] = maxval[i-1,j]

Verde: celle con valore già calcolato 🟩
Giallo: celle con valore da calcolare 🟨
Rosso: celle di confronto 🟥

	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

Oggetto 1 🟥 (i=1) valore 10, peso 8



i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	10	0	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0

maxval

j = 8

preso/non preso

items[i-1].size $\leq$ j $8 \leq 8 \Rightarrow$ l’oggetto 1 può essere preso

conveniente/non conveniente

maxval[i-1,j] < maxval[i-1][j-items[i-1].size]+items[i-1].value)

0 0 10

$\Rightarrow$ l’oggetto 1 conviene e viene preso maxval[i,j] = 10

Verde: celle con valore già calcolato 🟩
Giallo: celle con valore da calcolare 🟨
Rosso: celle di confronto 🟥

	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

Oggetto 1 🟥 (i=1) valore 10, peso 8

i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	10	10	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0

maxval

j = 9

preso/non preso

items[i-1].size ≤ j 8 ≤ 9 ⇒ l'oggetto 1 può essere preso

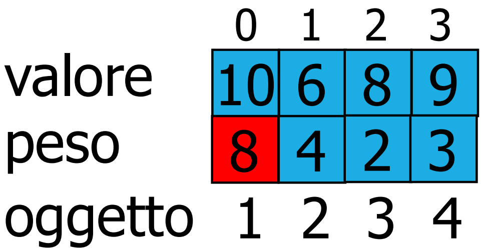
conveniente/non conveniente

maxval[i-1,j] < maxval[i-1][j-items[i-1].size]+items[i-1].value)

0 0 10

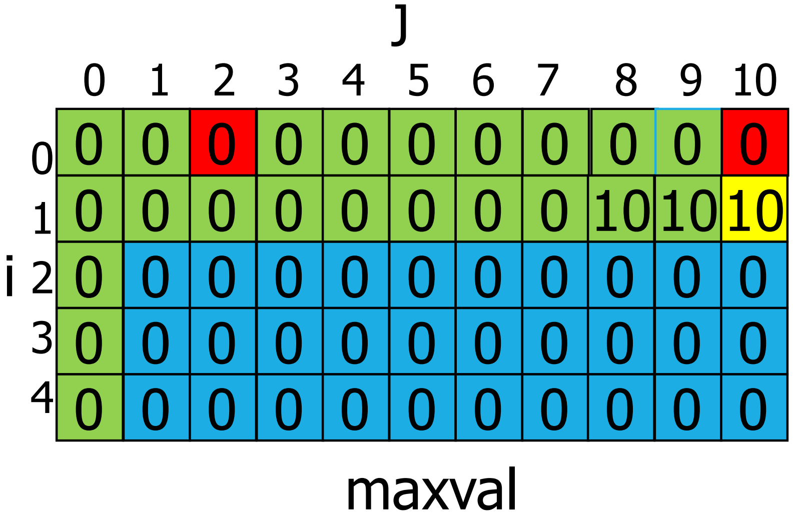
⇒ l'oggetto 1 conviene e viene preso maxval[i,j] = 10

Verde: celle con valore già calcolato 🟩
Giallo: celle con valore da calcolare 🟨
Rosso: celle di confronto 🟥



	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

Oggetto 1 🟥 (i=1) valore 10, peso 8



i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	10	10	10
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0

maxval

j = 10

preso/non preso

items[i-1].size ≤ j 8 ≤ 10 ⇒ l'oggetto 1 può essere preso

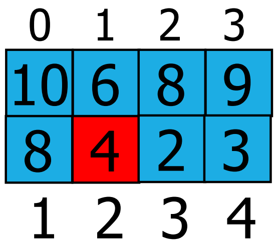
conveniente/non conveniente

maxval[i-1,j] < maxval[i-1][j-items[i-1].size]+items[i-1].value)

0 0 10

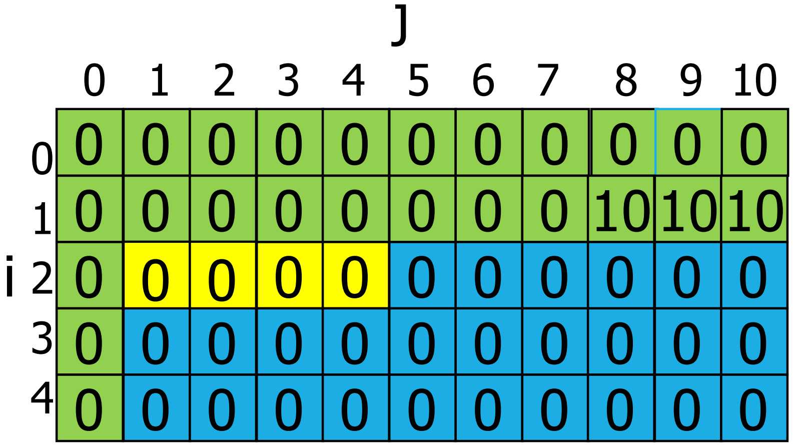
⇒ l'oggetto 1 conviene e viene preso maxval[i,j] = 10

Verde: celle con valore già calcolato 🟩
Giallo: celle con valore da calcolare 🟨
Rosso: celle di confronto 🟥



	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

Oggetto 2 🟥 (i=2) valore 6, peso 4



i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	10	10	10
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0

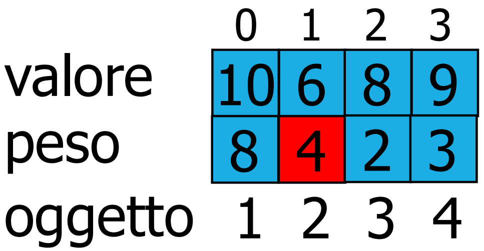
maxval

$\forall j \; 1 \leq j < 4$

items[i-1].size > j ⇒ l’oggetto 2 non viene preso

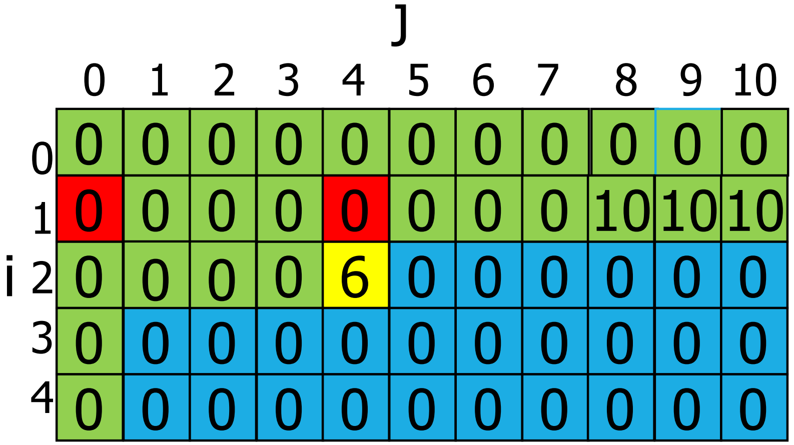
⇒ maxval[i, j] = maxval[i-1,j]

Verde: celle con valore già calcolato 🟩
Giallo: celle con valore da calcolare 🟨
Rosso: celle di confronto 🟥



	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

Oggetto 2 🟥 (i=2) valore 6, peso 4



i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	10	10	10
2	0	0	0	0	6	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0

maxval

j = 4

preso/non preso

items[i-1].size ≤ j 4 ≤ 4 ⇒ l'oggetto 2 può essere preso

conveniente/non conveniente

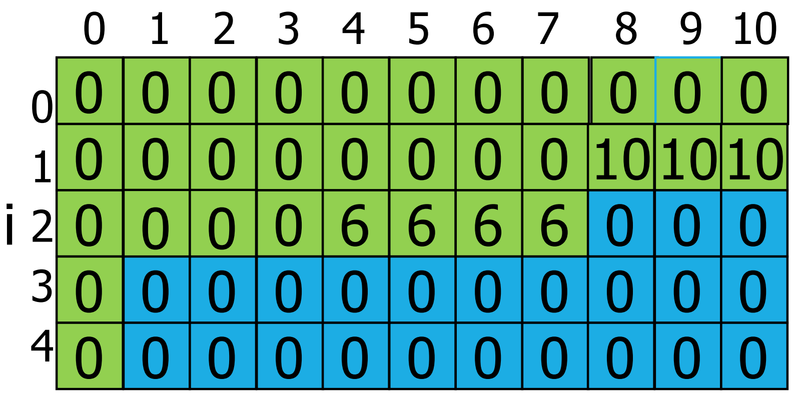
$$\underset{0}{\text{maxval[i-1,j]}} < \underset{0}{\text{maxval[i-1][j-items[i-1].size]}}+\underset{6}{\text{items[i-1].value)}}$$

⇒ l'oggetto 2 conviene e viene preso maxval[i,j] = 6

Verde: celle con valore già calcolato 🟩
Giallo: celle con valore da calcolare 🟨
Rosso: celle di confronto 🟥

	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

Oggetto 2 🟥 (i=2) valore 6, peso 4



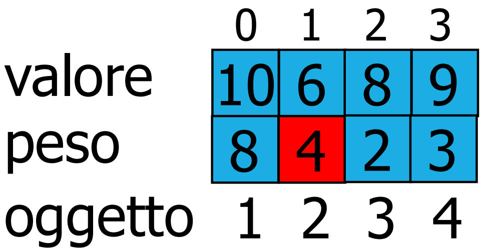
j

		0	1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0	0	10	10	10
i	2	0	0	0	0	6	6	6	6	0	0	0
	3	0	0	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0	0	0

maxval

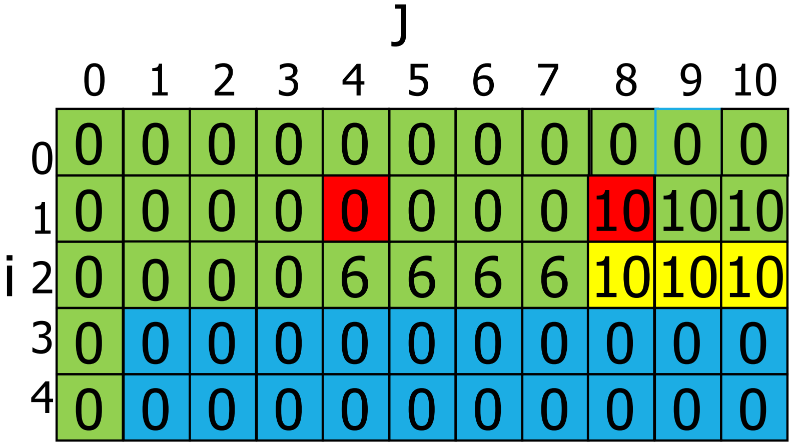
Analogamente per j = 5, 6 e 7
preso/non preso: l’oggetto 2 può essere preso
conveniente/non conveniente: l’oggetto 2 conviene, preso

Verde: celle con valore già calcolato 🟩
Giallo: celle con valore da calcolare 🟨
Rosso: celle di confronto 🟥



	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

Oggetto 2 🟥 (i=2) valore 6, peso 4



j

i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	10	10	10
2	0	0	0	0	6	6	6	6	10	10	10
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0

maxval

j = 8

items[i-1].size < 8 ⇒ l'oggetto 2 può essere preso conveniente/non conveniente

$$\underset{10}{\text{maxval[i-1,j]}} > \underset{0}{\text{maxval[i-1][j-items[i-1].size]}}+\underset{6}{\text{items[i-1].value}})$$

⇒ l'oggetto 2 non conviene, quindi non viene preso e maxval[i,j] = maxval[i-1,j] = 10

Analogamente per j = 9 e j = 10.

Si procede allo stesso modo per i = 3 e i = 4.

	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

N = 4

cap = 10

j

maxval =

i

0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	10	10	10
0	0	0	0	6	6	6	6	10	10	10
0	0	8	8	8	8	14	14	14	14	18
0	0	8	9	9	17	17	17	17	23	23

configurazione finale

Soluzione ottima: costruzione

Iterazione: partendo da maxval[N][cap] si scandiscono tutti gli oggetti:

- se la stima maxval[i][j] che include l'oggetto corrente i è uguale a quella che non lo include maxval[i-1][j] significa che l'oggetto corrente i non è stato preso, quindi sol[i-1] = 0
- altrimenti l'oggetto corrente i è stato preso (sol[i-1]=1). L'oggetto preso ha «consumato» una certa capacità, quindi j viene aggiornato a j – items[i-1].size.

Si ricordi che l'oggetto corrente i si trova in items[i-1] e il suo flag di presenza/assenza in sol[i-1].

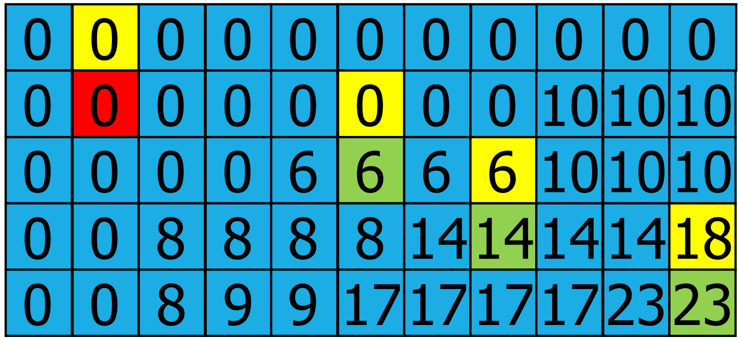
	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

	1	2	3	4
sol	0	1	1	1

j

maxval =

i



0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	10	10	10
0	0	0	0	6	6	6	6	10	10	10
0	0	8	8	8	8	14	14	14	14	18
0	0	8	9	9	17	17	17	17	23	23

Soluzione:
insieme {B, C, D} con valore massimo 23

```
void displaySol(Item *items, int N, int cap, int **maxval){
  int i, j, *sol;
  sol = calloc(N, sizeof(int));
  j = cap;
  for (i=N; i>=1; i--)
    if (maxval[i][j] == maxval[i-1][j])
      sol[i-1] = 0;
    else  {
      sol[i-1] = 1;
      j-= items[i-1].size;
    }
  for (i=0; i<N; i++)
    if (sol[i])
      ITEMstore(items[i]);
}
```

oggetto non preso

oggetto preso

visualizzazione degli oggetti presi

Complessità

KNAPmaxValDP: $T(n) = \Theta(N \cdot cap)$: conta solo il numero di sottoproblemi, in quanto ognuno ha costo $\Theta(1)$.

Memoization

- Approccio simile al divide et impera ricorsivo, quindi top-down
- Memorizzazione delle soluzioni ai sottoproblemi
- Lookup: evita di risolvere problemi già trattati
- NB: alcuni autori la denominano Programmazione dinamica top-down

Esempio: numeri di Fibonacci

array knownF

```
unsigned long fib(int n, unsigned long *knownF) {
  unsigned long t;
  if (knownF[n] != -1)
    return knownF[n];
  if(n == 0 || n == 1) {
    knownF[n] = n;
    return n;
  }
  t = fib(n-2, knownF) + fib(n-1, knownF);
  knownF[n] = t;
  return t;
}
```

07fibonacciM

Esempio: sequenza di Hofstadter

$H(0)=0$

$H(n)=n-H(H(H(n-1)))$ per $n > 0$

```
int H(int n) {
  if (n == 0)
    return 0;
  return n - H(H(H(n-1)));
}
```

ricorsivo

```
typedef struct hm_ {int val;int calc;} hm;
int HM(int n, hm *hmem) {
  if (n == 0) {
    hmem[n].val  = 0; hmem[n].calc = 1;
    return hmem[n].val;
  }
  if (hmem[n].calc == 1) return hmem[n].val;
  hmem[n].val=n-HM(HM(HM(n-1,hmem),hmem),hmem);
  hmem[n].calc = 1;
  return hmem[n].val;
}
```

array hmem di struct

memoization

08hofstadterM

Esempio: parentesizzazione ottima

```
int matrix_chainM(int *p, int n) {
  int i, j, **m;
  m = malloc((n+1)*sizeof(int *));
  for (i = 0; i <= n; i++)
    m[i] = malloc((n+1)*sizeof(int));
  for (i = 0; i <= n; i++)
    for (j = 0; j <= n; j++)
      m[i][j] = INT_MAX;
  return lookup(p, 1, n, m);
}
```

matrice m

Complessità $T(n) = O(n^3)$

$S(n) = \Theta(n^2)$

03matrix_chain_mult

```
int lookup(int *p, int i, int j, int **m) {
  int k, q;
  if (m[i][j] < INT_MAX)
    return m[i][j];
  if (i==j)
    m[i][j] = 0;
  else
    for (k= i; k <j; k++) {
      q = lookup(p, i, k, m) +
          lookup(p, k+1, j, m) + p[i-1]*p[k]*p[j];
      if (q < m[i][j])
        m[i][j] = q;
    }
  return m[i][j];
}
```

Esempio: il problema dello zaino

Dato un insieme di N oggetti ciascuno dotato di peso w_j e di valore v_j e dato un peso massimo cap, determinare il sottoinsieme S di oggetti tali che:

- $\sum_{j \in S} w_j x_j \leq$ cap
- $\sum_{j \in S} v_j x_j =$ MAX
- $x_j \in \{0,1\}$

Ogni oggetto o è preso (x_j =1) o lasciato (x_j=0). Ogni oggetto esiste in un numero arbitrario di istanze.

Esempio

N = 5 cap = 17

		A	B	C	D	E
Valore	v_i	4	5	10	11	13
Peso	p_i	3	4	7	8	9

Possibili soluzioni:

- D, E: peso 17, valore 24
- A, C, C: peso 17, valore 24

soluzione ottima!

- A, A, B, C: peso 17, valore 23
- A, A, A, B, B: peso 17, valore 22

soluzione non ottima!

Soluzione ricorsiva: calcolo del valore

- principio di moltiplicazione: le scelte sono gli oggetti (`i`)
- l’oggetto `i` può essere scelto se il suo peso è compatibile con la capacità disponibile, cioè se la capacità disponibile space dopo aver preso l’oggetto è ≥ 0

```
space = cap-items[i].size >= 0;
```

- se scelto, si ricalcola la capacità disponibile

```
space = cap-items[i].size);
```

- ricorsione: determinazione della soluzione ottima per zaino di capacità disponibile `space`
- se si migliora il risultato corrente, lo si aggiorna

```
if((t=knapR(space)+items[i].value) > max)
  max = t;
```

```
int KNAPmaxValR(Item *items, int N, int cap) {
  int i, space, max, t;
  for (i = 0, max = 0; i < N; i++)
    if ((space = cap-items[i].size) >= 0)
      if ((t=KNAPmaxValR(items,N,space)+items[i].value)>max)
        max = t;
  return max;
}
```

ricorsivo

09knapsackM

```
int KNAPmaxValM(Item *items,int N,int cap,int *maxKnown){
  int i, space, max, t;
  if (maxKnown[cap] != -1)
    return maxKnown[cap];
  for (i=0, max=0; i < N; i++)
    if ((space = cap-items[i].size) >= 0)
      if ((t=KNAPmaxValM(items,N,space,maxKnown)+
             items[i].value)>max)
        max = t;
  maxKnown[cap] = max;
  return max;
}
```

memoization

Riferimenti

- Programmazione dinamica:
    - Cormen 16
    - Montresor 13
    - Crescenzi 2.7
- Problema del ladro e dello zaino:
    - Sedgewick 5.3

Esercizi di teoria

- 8. Programmazione dinamica
  - 8.1 Prodotto in catena di matrici
  - 8.2 Longest Common Subsequence

Approfondimenti

- Longest Common Subsequence