

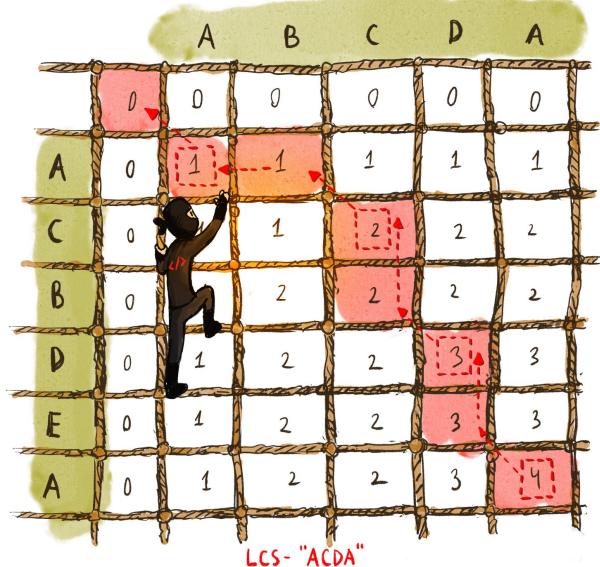
Paolo
Ferragina
lunedì 17-19

Programmazione dinamica

Giuseppe Prencipe
giuseppe.prenclipe@unipi.it

Paradigma della programmazione dinamica

- Altro **paradigma fondamentale**, come il *divide et impera*
- Calcolo efficiente di una funzione ricorsiva mediante la **memorizzazione** dei suoi **valori intermedi** in una **tabella** (detta di *programmazione dinamica*)



$n \geq 0$

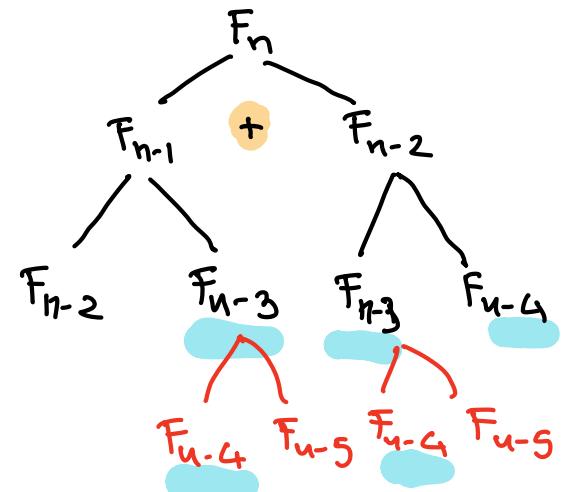
Fib(n) // $n \geq 0$

if $n \leq 1$ return n
return Fib(n-1) + Fib(n-2)

Programmazione dinamica: Fibonacci

Esempio: numeri di Fibonacci (0, 1, 1, 2, 3, 5, 8, 13,...)

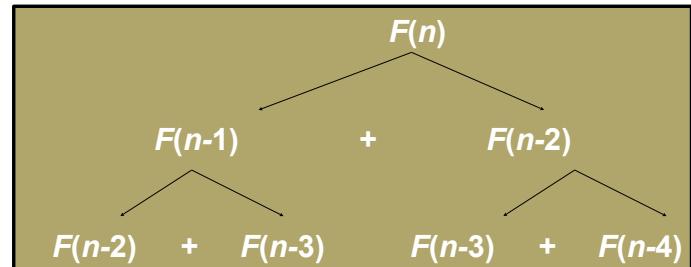
$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad \text{per } n \geq 2 \end{cases}$$

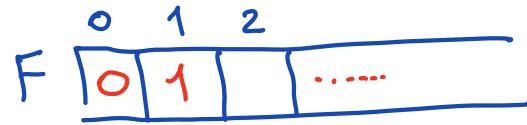


Soluzione ricorsiva

```
Fib( n ) {           // n >= 0
    if (n <= 1)
        RETURN n;
    else
        RETURN Fib(n-1) + Fib(n-2);
}
```

- Approccio **top-down**
- $\text{Fib}(n)$ richiede $O(F_n) = O(\phi^n)$ passi,
dove $\phi = (1 + \sqrt{5})/2 = 1,6180339\dots$ è il **rapporto aureo**

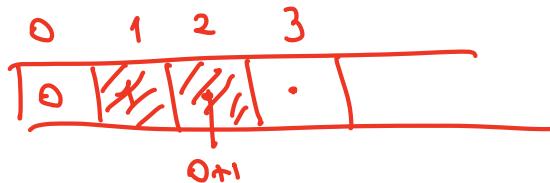




Fibonacci con programmazione dinamica

```

Fib( n ) {           // n >= 0
    F[0] = 0;          | array pre-allocato
    F[1] = 1;
    for (k = 2; k <= n; k = k + 1)
        F[k] = F[k-1] + F[k-2];
    RETURN F[n];
}
  
```



- Costo:
 - Tempo – O(n)
 - Spazio – O(n)

O(\downarrow)

- L'array F è la **tabella di programmazione dinamica**
- **Idea di base:** F[k] si calcola utilizzando valori di F già calcolati
- **RISPARMIO:**

Programmazione Dinamica

- La Programmazione Dinamica è una tecnica di progettazione di algoritmi utilizzata con ***problemi di ottimizzazione***
- Come il *divide et impera*, risolve problemi combinando soluzioni di sotto-problemi
- A differenza del *divide et impera*, i sotto-problemi non sono indipendenti
 - Sotto-problemi (tipicamente) condividono sotto-problemi

Programmazione Dinamica vs Divide et impera

- Divide et impera
 - Partiziona il problema in sotto-problemi **indipendenti**
 - ✓ Risolvi i sotto-problemi ricorsivamente
 - ✓ Combina le soluzioni per risolvere il problema originale

 - Programmazione Dinamica
 - Partiziona il problema in sotto-problemi **(non indipendenti)**
 - Risolvi i sotto-problemi ricorsivamente
 - Combina le soluzioni per risolvere il problema originale
- 
- Tabella

Programmazione Dinamica



Richard Ernest Bellman (August 26, 1920 – March 19, 1984) was an American applied mathematician, who introduced dynamic programming in 1953, and made important contributions in other fields of mathematics.

- Il termine **Dynamic Programming** viene dalla *Teoria dei Controlli*, non dall'Informatica
 - Termine introdotto negli anni '50 da Richard Bellman, che sviluppò metodi di **Programmazione** logistica per l'aeronautica.
- **Dinamica** si riferisce al fatto che la tabella è riempita progressivamente
- **Programming** si riferisce all'utilizzo di tabelle (array) per costruire una soluzione

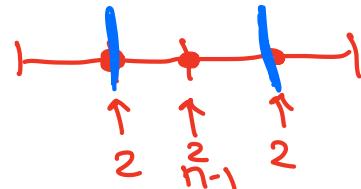
Programmazione Dinamica

- Il modo migliore per comprendere questa tecnica è attraverso esempi — ragionamenti simili (ma leggermente diversi) per la stessa conclusione
 - Taglio della corda ✓
 - Calcolo coefficienti binomiali ✓
 - Matrix Chaining optimization (ex)
 - Longest Common Subsequence ✓ (edir di source)
 - Problema dello Zaino 0-1

Taglio della corda (rod-cutting)

Taglio della corda (rod-cutting)

- Ogni pezzo di corda ha un prezzo \leftrightarrow guadagno
- Massimizzare il guadagno!

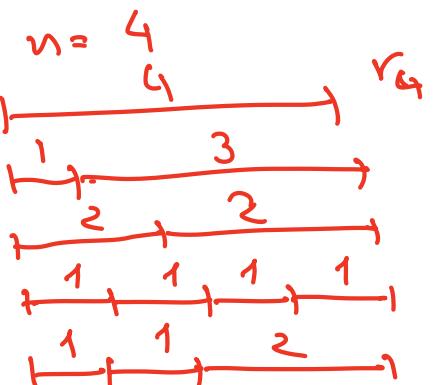


$O(n)$

Possibili tagli?

$P_1 + P_3$

$P_2 + P_2$



Taglio della corda (rod-cutting)

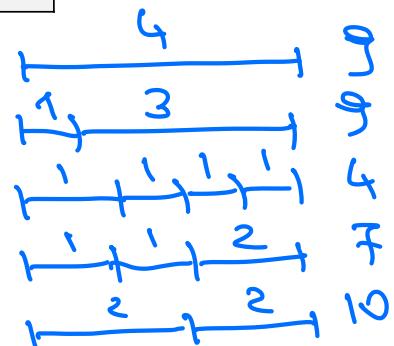
Definizione del problema

- Data una corda di lunghezza n centimetri e una tabella di prezzi p_i , $i=1,2,\dots,n$, trovare il massimo guadagno r_n ottenibile tagliando la corda e vendendo i singoli pezzi \equiv
 - Lunghezze della corda sono interi
 - Per $i=1,2,\dots,n$ conosciamo il prezzo p_i di un pezzo di corda lungo i cm

Taglio della corda: esempio

lunghezza	1	2	3	4	5	6	7	8	9	10
prezzo p_i	1	5	8	9	10	17	17	20	24	30

- Per una corda di **lunghezza 4**: quale taglio è ottimale?



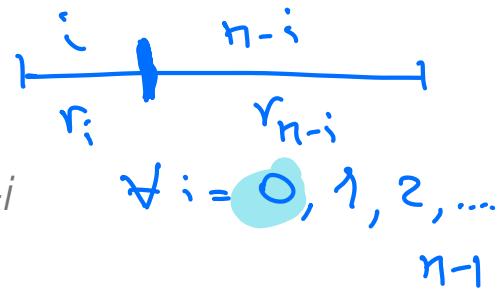
Taglio della corda: esempio

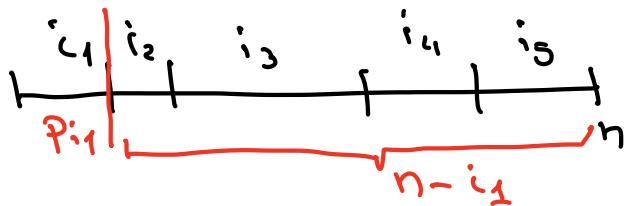
lunghezza l	1	2	3	4	5	6	7	8	9	10
prezzo p_i	1	5	8	9	10	17	17	20	24	30

- Per una corda di lunghezza 4: 2+2 è ottimale ($p_2+p_2=10$)
- In generale, ci sono 2^{n-1} modi per tagliare una corda di lunghezza n!

Taglio della corda

- Se la soluzione ottima taglia la corda in k pezzi, allora
 - Decomposizione ottima: $n = i_1 + i_2 + \dots + i_k$
 - Guadagno: $r_n = p_{i_1} + p_{i_2} + \dots + p_{i_k}$
- Taglio **iniziale** della corda: due pezzi lunghi i e $n-i$
 - Guadagno r_i e r_{n-i} derivati da questi due pezzi
 - Bisogna considerare tutti i possibili valori di i
 - Ovviamente va considerato anche il caso in cui la corda è venduta senza tagli
- In altre parole: $r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1\}$





Una differente visione del problema

- Decomponiamo in
 - Un primo pezzo, a sx, di lunghezza i
 - Il secondo pezzo, che resta a dx, di lunghezza $n-i$
 - Si prova a dividere ulteriormente solo il pezzo a dx
 - Questo per ogni i
 - Quindi
 - $r_n = \max\{p_i + r_{n-i}, 1 \leq i \leq n\}$
 - Cerchiamo la soluzione solo a un sotto-problema

ricorsione
di
coda

$$n = 4$$

Taglio della corda: esempio

Corda da 4 **Taglia(4)**

i = 1 **Taglia(4)** Costo????



$p_1 + r_{4-1}$
↳ Taglia (3)

- $r_n = \max\{p_i + r_{n-i}, 1 \leq i \leq n\}$

$$p[] = [1, 5, 8, 9]$$

Taglio della corda: esempio

$$\cdot r_n = \max\{p_i + r_{n-i}, 1 \leq i \leq n\}$$

Corda da 4 **Taglia(4)**

$$p[] = [1, 5, 8, 9]$$

i = 1 **Taglia(4)** Costo: $p_1 + \text{Taglia}(3)$

Ricorsivamente, ripartiamo da $i=1$

i = 1 **Taglia(3)** Costo????

Taglio della corda: esempio

$$\cdot r_n = \max\{p_i + r_{n-i}, 1 \leq i \leq n\}$$

Corda da 4 **Taglia(4)**

$$p[] = [1, 5, 8, 9]$$

i = 1 **Taglia(4)**

$$\text{Costo: } p_1 + \text{Taglia}(3)$$

Ricorsivamente, ripartiamo da i=1

i = 1

Taglia(3)

$$\text{Costo: } p_1 + \text{Taglia}(2)$$

Ricorsivamente, ripartiamo da i=1

i = 1

Taglia(2)

$$\text{Costo: } p_1 + \text{Taglia}(1) = 1 + 1 = 2$$

i = 2

Taglia(2)

$$\text{Costo: } p_2 + \text{Taglia}(0) = 5 + 0 = 5$$

ORA????

5

Taglio della corda: esempio

$$\cdot r_n = \max\{p_i + r_{n-i}, 1 \leq i \leq n\}$$

Corda da 4 **Taglia(4)**

$$p[] = [1, 5, 8, 9]$$

i = 1 **Taglia(4)** Costo: $p_1 + \text{Taglia}(3)$

Ricorsivamente, ripartiamo da $i=1$

i = 1

Taglia(3) Costo: $p_1 + \text{Taglia}(2)$

Ricorsivamente, ripartiamo da $i=1$

i = 1

Taglia(2) Costo: $p_1 + \text{Taglia}(1) = 1 + 1 = 2$

i = 2

Taglia(2) Costo: $p_2 + \text{Taglia}(0) = 5 + 0 = 5$

→ Ottimo Taglia(2) è ????

Taglio della corda: esempio

$$\cdot r_n = \max\{p_i + r_{n-i}, 1 \leq i \leq n\}$$

Corda da 4 **Taglia(4)**

$$p[] = [1, 5, 8, 9]$$

i = 1 **Taglia(4)**

Costo: $p_1 + \text{Taglia}(3)$

Ricorsivamente, ripartiamo da $i=1$

i = 1

Taglia(3)

Costo: $p_1 + \text{Taglia}(2) = 1 + 5 = 6$

Ricorsivamente, ripartiamo da $i=1$

i = 1

Taglia(2)

Costo: $p_1 + \text{Taglia}(1) = 1 + 1 = 2$

Ottimo Taglia(2) è quindi 5

i = 2

Taglia(2)

Costo: $p_2 + \text{Taglia}(0) = 5 + 0 = 5$

Prosegue analisi di Taglia(3)

i = 2 per Taglia(3)

Taglia(3)

Costo: ????

$$p_2 + \text{Taglia}(1) = p_2 + p_1 = 5 + 1 = 6$$



Taglio della corda: esempio

$$\cdot r_n = \max\{p_i + r_{n-i}, 1 \leq i \leq n\}$$

Corda da 4 **Taglia(4)**

$$p[] = [1, 5, 8, 9]$$

i = 1 **Taglia(4)**

$$\text{Costo: } p_1 + \text{Taglia}(3)$$

Ricorsivamente, ripartiamo da i=1

i = 1

Taglia(3)

$$\text{Costo: } p_1 + \text{Taglia}(2) = 1 + 5 = 6$$

Ricorsivamente, ripartiamo da i=1

i = 1

Taglia(2)

$$\text{Costo: } p_1 + \text{Taglia}(1) = 1 + 1 = 2$$

Ottimo Taglia(2) è quindi 5

i = 2

Taglia(2)

$$\text{Costo: } p_2 + \text{Taglia}(0) = 5 + 0 = 5$$

Prosegue analisi di Taglia(3)

i = 2 per Taglia(3)

Taglia(3)

$$\text{Costo: } p_2 + \text{Taglia}(1) = 5 + 1 = 6$$

i = 3 per Taglia(3)

Taglia(3)

$$\text{Costo: } ????$$

Taglio della corda: esempio

$$\cdot r_n = \max\{p_i + r_{n-i}, 1 \leq i \leq n\}$$

Corda da 4 **Taglia(4)**

$$p[] = [1, 5, 8, 9]$$

i = 1 **Taglia(4)**

Costo: $p_1 + \text{Taglia}(3)$

Ricorsivamente, ripartiamo da $i=1$

i = 1

Taglia(3)

Costo: $p_1 + \text{Taglia}(2) = 1 + 5 = 6$

Ricorsivamente, ripartiamo da $i=1$

i = 1

Taglia(2)

Costo: $p_1 + \text{Taglia}(1) = 1 + 1 = 2$

→ Ottimo Taglia(2) è quindi 5

i = 2

Taglia(2)

Costo: $p_2 + \text{Taglia}(0) = 5 + 0 = 5$

Prosegue analisi di Taglia(3)

i = 2 per Taglia(3)

Taglia(3)

Costo: $p_2 + \text{Taglia}(1) = 5 + 1 = 6$

→ Ottimo Taglia(3) è ????

i = 3 per Taglia(3)

Taglia(3)

Costo: $p_3 + \text{Taglia}(0) = 8 + 0 = 8$

Taglio della corda: esempio

Corda da 4

Taglia(4)

$$r_n = \max\{p_i + r_{n-i}, 1 \leq i \leq n\}$$

$p[] = [1, 5, 8, 9]$

i = 1

Taglia(4)

Costo: $p_1 + \text{Taglia}(3)$

Ricorsivamente, ripartiamo da $i=1$

i = 1

Taglia(3)

Costo: $p_1 + \text{Taglia}(2) = 1 + 5 = 6$

Ricorsivamente, ripartiamo da $i=1$

i = 1

Taglia(2)

Costo: $p_1 + \text{Taglia}(1) = 1 + 1 = 2$

→ Ottimo Taglia(2) è quindi 5

i = 2

Taglia(2)

Costo: $p_2 + \text{Taglia}(0) = 5 + 0 = 5$

Prosegue analisi di Taglia(3)

i = 2 per Taglia(3)

Taglia(3)

Costo: $p_2 + \text{Taglia}(1) = 5 + 1 = 6$

→ Ottimo Taglia(3) è quindi 8

i = 3 per Taglia(3)

Taglia(3)

Costo: $p_3 + \text{Taglia}(0) = 8 + 0 = 8$

E così via....

TAGLIA

TAGLIA

n	i	p [i]	CUT-ROD (p, n-i)	p [i] + CUT-ROD (p, n-i)	max
4	1 2 3 4	1 5 8 9	(p, 4 - 1) = (p, 3) = 8 (p, 4 - 2) = (p, 2) = 5 (p, 4 - 3) = (p, 1) = 1 (p, 4 - 4) = (p, 0) = 0	1 + 8 = 9 5 + 5 = 10 8 + 1 = 9 9 + 0 = 9	
3	1 2 3	1 5 8	(p, 3 - 1) = (p, 2) = 5 (p, 3 - 2) = (p, 1) = 1 (p, 3 - 3) = (p, 0) = 0	1 + 5 = 6 5 + 1 = 6 8 + 0 = 8	8
2	1 2	1 5	(p, 2 - 1) = (p, 1) = 1 (p, 2 - 2) = (p, 0) = 0	1 + 1 = 2 5 + 0 = 5	5
1	1	1	(p, 1 - 1) = (p, 0) = 0	1 + 0 = 1	1

Implementazione top-down

TAGLIO(p,n)

// TAGLIA

$$r_n = \max\{p_i + r_{n-i}, 1 \leq i \leq n\}$$

$\text{D}(i)$

```
if n==0
    return 0
    q = -∞
for i=1 to n
    •  $q = \max\{q, p[i] + \text{TAGLIO}(p, n-i)\}$ 
return q
```

$T(n-i)$

$\left. \begin{array}{l} \\ \\ \end{array} \right\} \text{calcolo MAX}$

- Tempo: $T(n) = 1 + ????$

$$\overbrace{\quad}^{\text{1}} - T(n-1) + \underbrace{T(n-2) + \dots + T(1)}_{\text{n-1}}$$

Implementazione top-down

TAGLIO(p,n)

if $n==0$

 return 0

q = $-\infty$

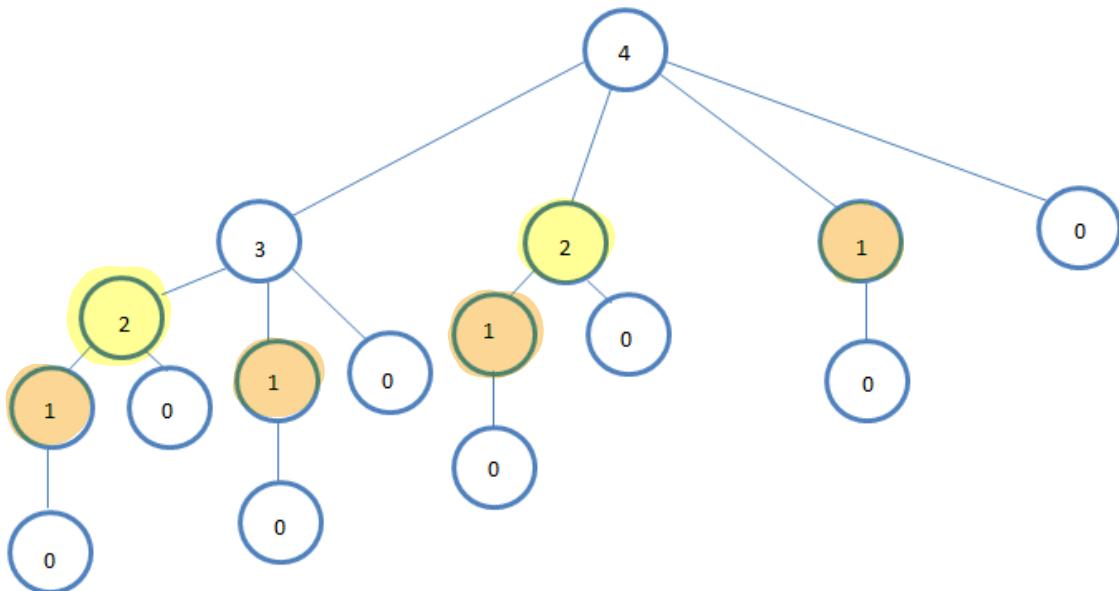
for $i=1$ to n

$q=\max\{q, p[i]+TAGLIO(p,n-i)\}$

return q

- Tempo: $T(n)=1+T(1)+T(2)+\dots+T(n-1)$
- $T(n)=O(2^n)$ *dim per induzione*

Implementazione top-down



Albero ricorsione di TAGLIO(p, n), $n = 4$.

Programmazione Dinamica –**bottom-up**

L'idea, come fatto per Fibonacci, è partire dai **valori che già conosciamo**, che sono quelli più piccoli

Su quelli più piccoli, che memorizziamo nella **tabella di programmazione dinamica**, calcoliamo quelli via via più grandi, fino a trovare la soluzione del problema

Programmazione Dinamica – **bottom-up**



→ Ottimo = 1 = P_1

$$r = [0, 1, 0, 0, 0]$$

$\xrightarrow{j=1}$
 \underline{s}

Taglia(4)

$$p = \underline{\underline{s}} [1, 5, 8, 9]$$

Programmazione Dinamica – bottom-up

1



→ Ottimo = 1

$$r = [0, 1, 0, 0, 0]$$

Taglia(4)

j=1

$$p = [1, 5, 8, 9]$$

2



2 possibilità

$$\underbrace{1}_{\text{1}} \cdot \underbrace{1}_{\text{1}} = p_1 + r_1 = 1 + 1 = 2$$

$$\underbrace{2}_{\text{2}} \cdot \underbrace{0}_{\text{0}} = p_2 + r_0 = 5 + 0 = 5$$

Iterativamente, passiamo a j=2

\equiv

E ricordiamo anche la definizione generale del problema di ottimizzazione

$$r_n = \max\{p_i + r_{n-i}, 1 \leq i \leq n\}$$

Vanno sfruttati i valori ottimi calcolati per i sotto-problemi

Al momento conosciamo solo $r[1]$

Programmazione Dinamica –**bottom-up**

$$r = [0, 1, 0, 0, 0]$$

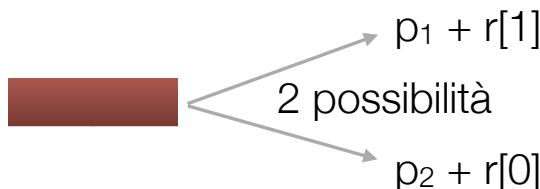
Taglia(4)

→ Ottimo = 1

j=1

p = [1, 5, 8, 9]

Iterativamente, passiamo a j=2



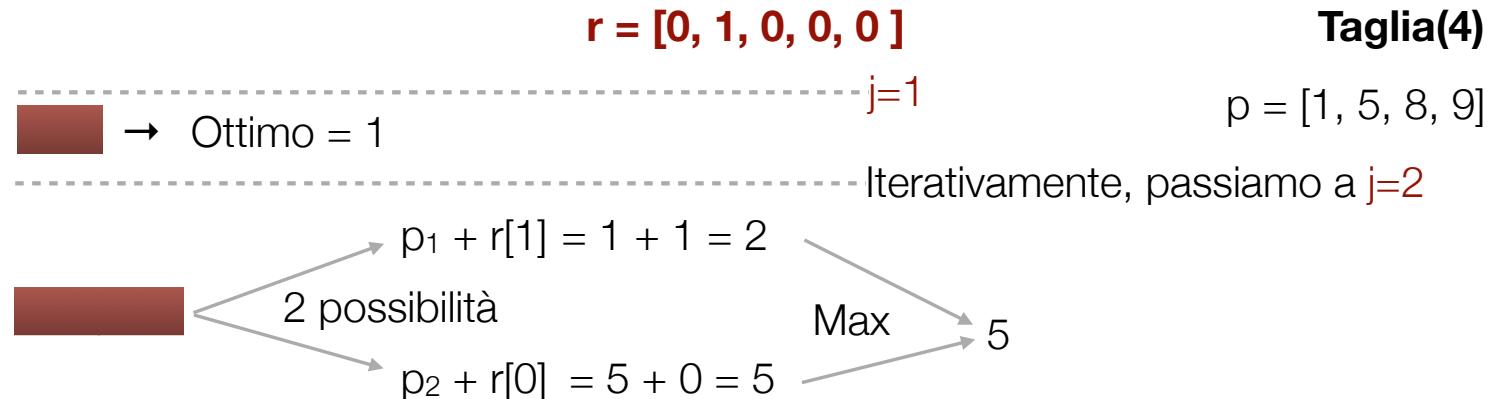
E ricordiamo anche la definizione generale del problema di ottimizzazione

$$r_n = \max\{p_i + r_{n-i}, 1 \leq i \leq n\}$$

Vanno sfruttati i valori ottimi calcolati per i sotto-problemi

Al momento conosciamo solo $r[1]$

Programmazione Dinamica –**bottom-up**



E ricordiamo anche la definizione generale del problema di ottimizzazione

$$r_n = \max\{p_i + r_{n-i}, 1 \leq i \leq n\}$$

Vanno sfruttati i valori ottimi calcolati per i sotto-problemi

Al momento conosciamo solo $r[1]$

Programmazione Dinamica – **bottom-up**

$$r = [0, 1, 5, 0, 0]$$

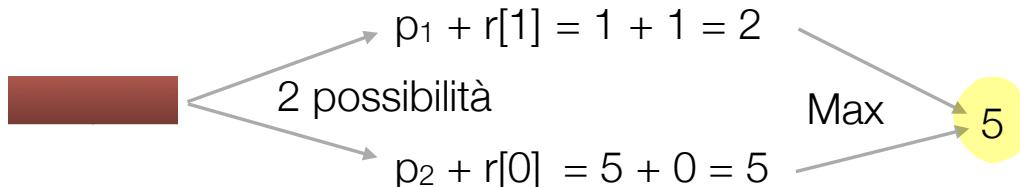
Taglia(4)

→ Ottimo = 1

j=1

$$p = [1, 5, 8, 9]$$

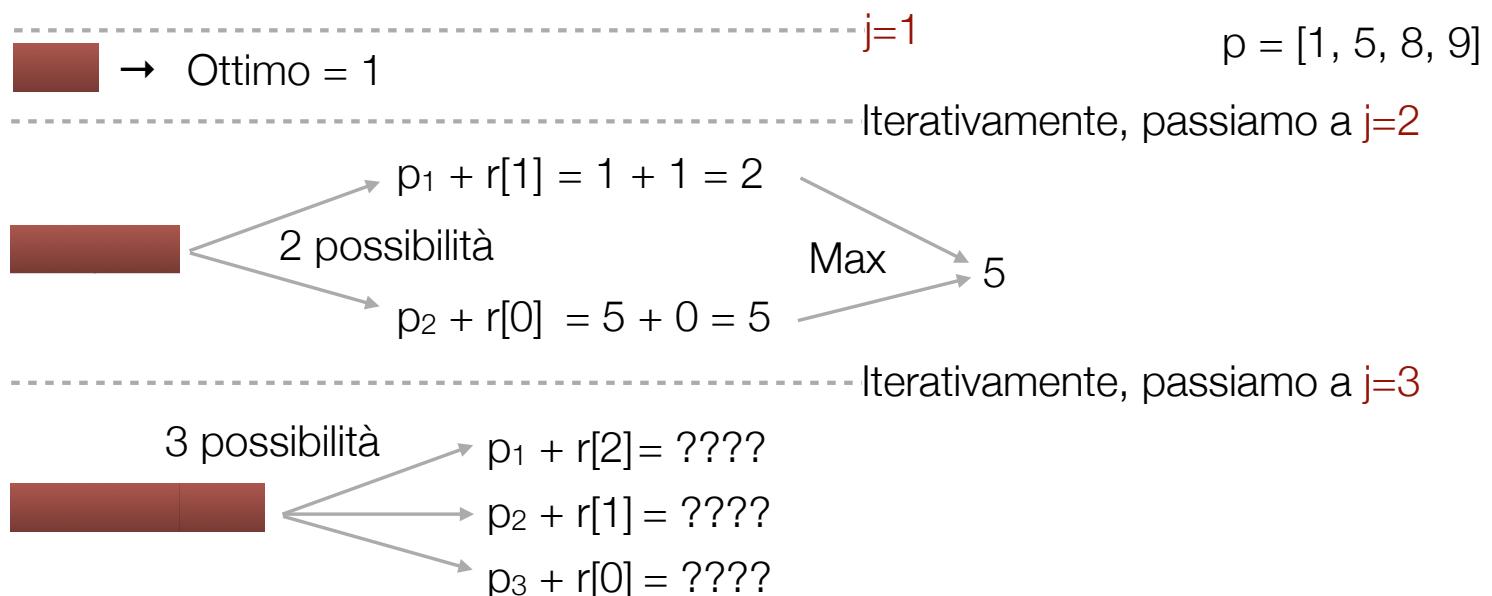
Iterativamente, passiamo a j=2



Programmazione Dinamica –**bottom-up**

$$r = [0, 1, 5, 0, 0]$$

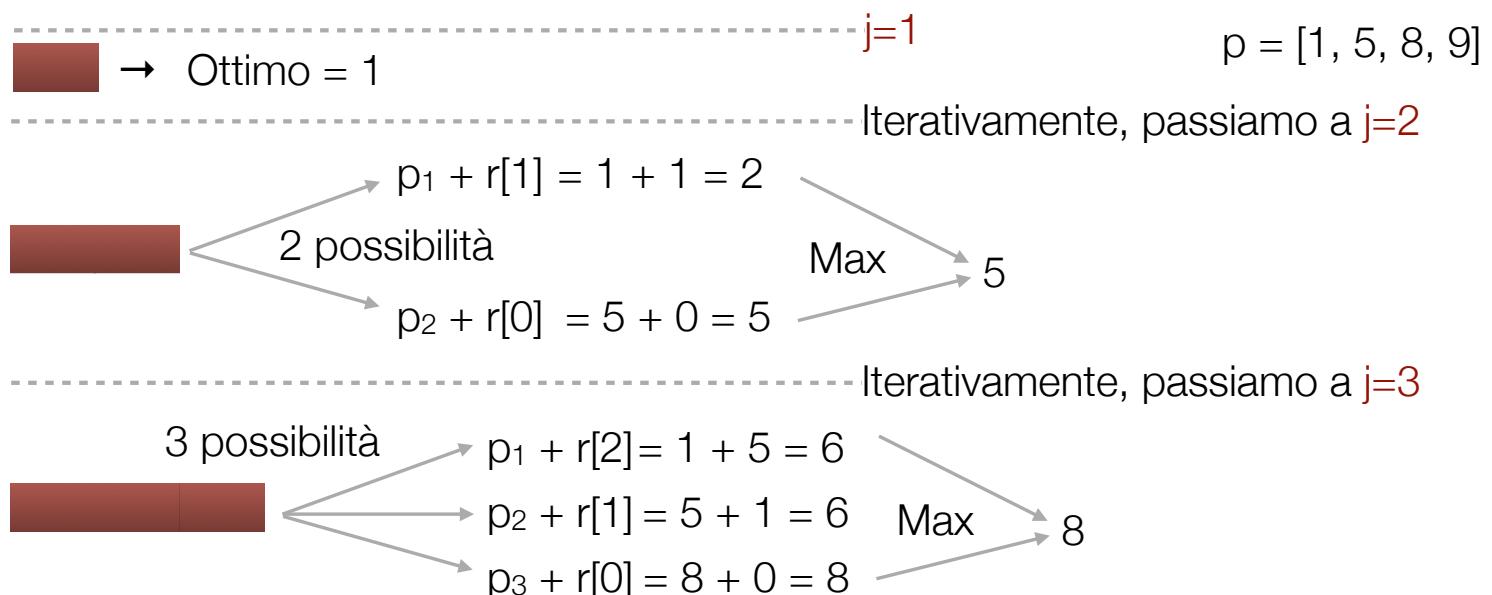
Taglia(4)



Programmazione Dinamica –**bottom-up**

$$r = [0, 1, 5, 0, 0]$$

Taglia(4)



Programmazione Dinamica – bottom-up

$$r = [0, 1, 5, 8, 0]$$

Taglia(4)

→ Ottimo = 1

j=1

$$p = [1, 5, 8, 9]$$

Iterativamente, passiamo a j=2

$$p_1 + r[1] = 1 + 1 = 2$$

2 possibilità

$$p_2 + r[0] = 5 + 0 = 5$$

Max

5

Iterativamente, passiamo a j=3

3 possibilità

$$p_1 + r[2] = 1 + 5 = 6$$

Max

8

$$p_2 + r[1] = 5 + 1 = 6$$

$$p_3 + r[0] = 8 + 0$$

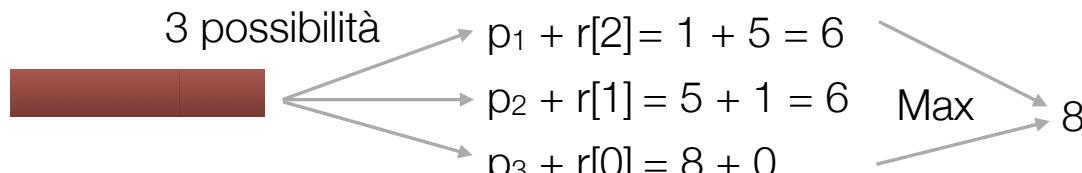
Programmazione Dinamica –**bottom-up**

$$r = [0, 1, 5, 8, 0]$$

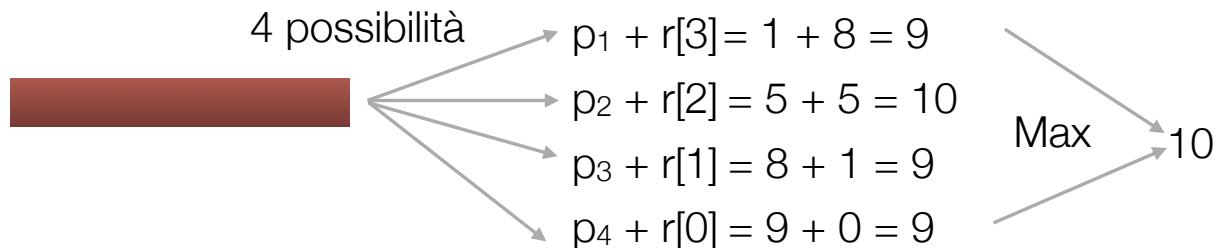
Taglia(4)

$$p = [1, 5, 8, 9]$$

Iterativamente, passiamo a **j=3**



Iterativamente, passiamo a **j=4**



Programmazione Dinamica – bottom-up

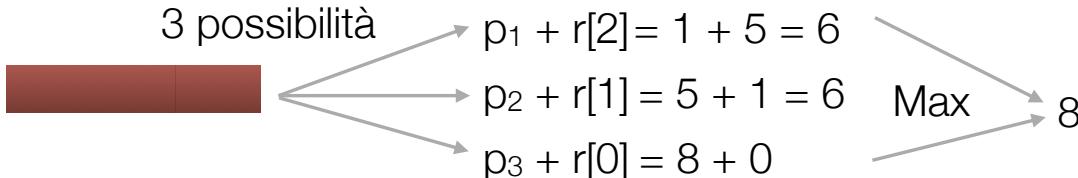
$$r = [0, 1, 5, 8, 10]$$

Taglia(4)

$$p = [1, 5, 8, 9]$$

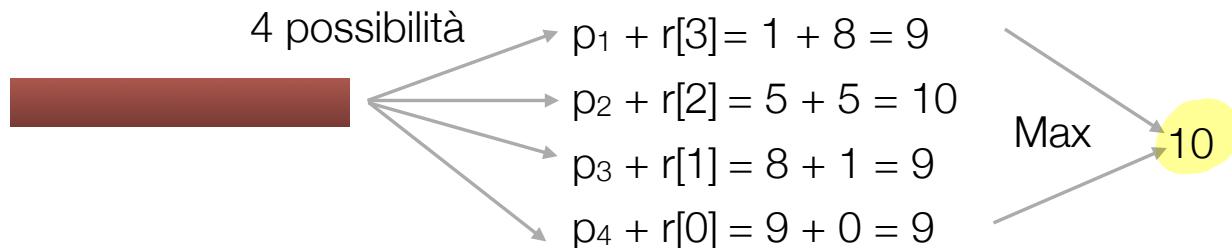
Iterativamente, passiamo a $j=3$

3 possibilità



Iterativamente, passiamo a $j=4$

4 possibilità



$$\sum_j O(j) \leq \sum_j c \cdot j = c \cdot \sum_j j$$

Programmazione Dinamica – **bottom-up**

TAGLIO-Dinamico(p, n)

$r[0..n] \leftarrow$

$r[0] = 0$

for $j=1$ to n

$q = -\infty$

for $i=1$ to j

if $q < p[i] + r[j-i]$

calcolo MAX

$O(j)$

$r[j] = q$

return r

E ricordiamo anche la definizione generale del problema di ottimizzazione

$$r_n = \max\{p_i + r_{n-i}, 1 \leq i \leq n\}$$

$$\sum_{j=1}^n O(j) = O\left(\sum_{j=1}^n j\right) = \Theta(n^2)$$

$$r_n = \max \{ p_i + r_{n-i} \}$$

$$\frac{n(n+1)}{2}$$

$$\underline{P_3} + \underline{r_{n-3}}$$

Recuperare la soluzione ottima

TAGLIO-Dinamico(p, n)

$r[0..n]$ and $s[0..n]$ arrays

$r[0]=0$

for $j=1$ to n

$q=-\infty$

for $i=1$ to j

if $q < p[i]+r[j-i]$

{ $s[j]=i$; $q=p[i]+r[j-i]$ }

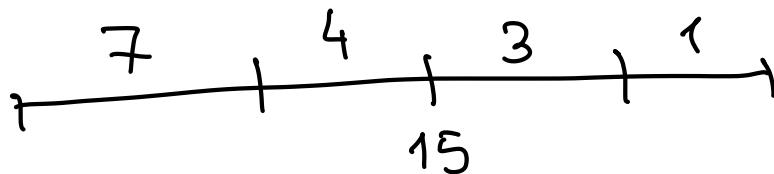
$r[j]=q$

return r and s

Ogni volta che troviamo un ottimo per un sotto-problema, dobbiamo ricordarci la posizione del taglio

Come modifichiamo l'algoritmo?

Utilizziamo un secondo array, $s[]$



Recuperare la soluzione ottima

- $r_n = \max\{p_i + r_{n-i}, 1 \leq i \leq n\}$

STAMPA-TAGLIO(n)

$$(r, s) = \text{TAGLIO-Dinamico}(p, n)$$

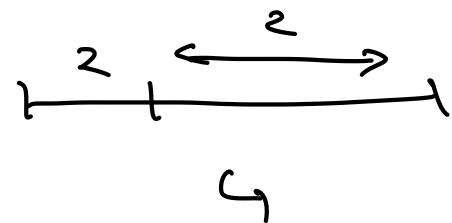
while $n > 0$

print $s[n]$

$n = n - s[n]$

Esempio:

$p = [1, 5, 8, 9]$	i	0	1	2	3	4
	$r[i]$	0	1	5	8	10
	$s[i]$	0	1	2	3	2



Coefficienti binomiali

Coefficienti binomiali

- Il coefficiente binomiale $C(n, k)$ è il numero di modi che si hanno per scegliere un sottoinsieme di k elementi da un insieme di n elementi
- Per definizione,

$$C(n, k) = \frac{n!}{k!(n - k)!}$$

- Questa formula non viene utilizzata per il calcolo effettivo, perché, anche per piccoli valori di n , **calcolare $n!$ è estremamente costoso**
- Si utilizza invece la seguente formula per il calcolo di $C(n,k)$

$$C(n,0)=1$$

$$C(n,n)=1$$

$$C(n,k)=C(n-1, k-1)+C(n-1, k)$$

Coefficienti binomiali – top-down

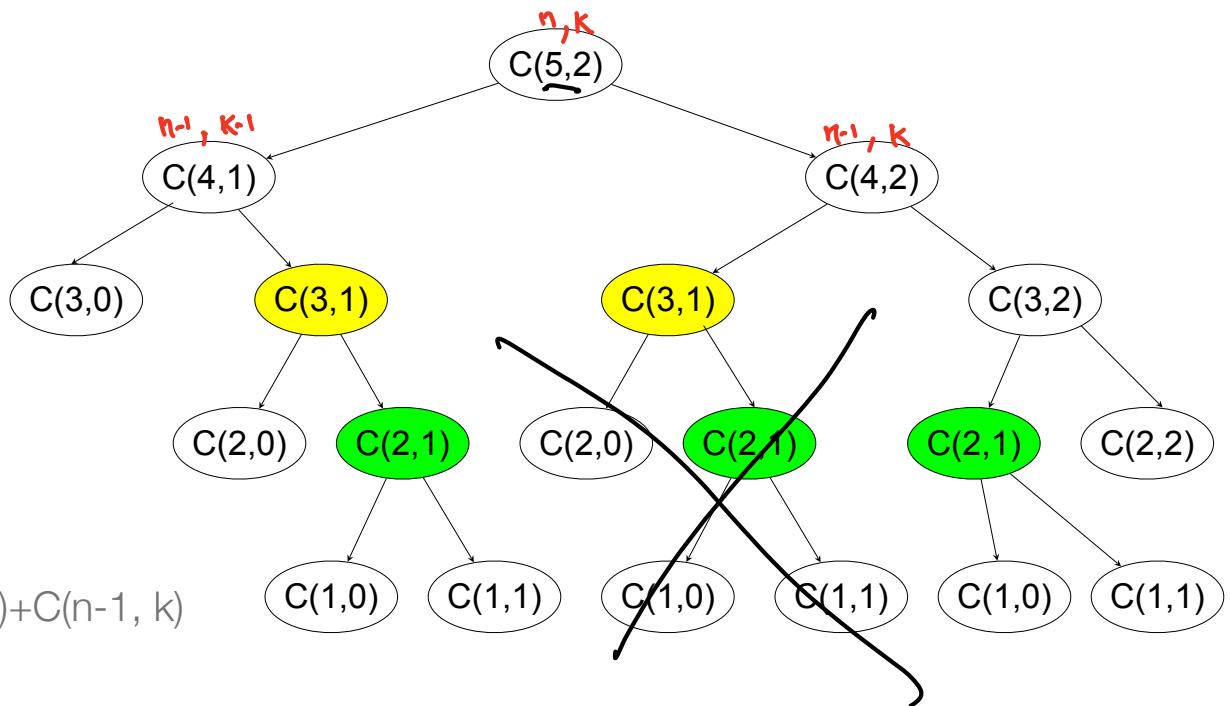
$$C(n,0)=1$$

$$C(n,n)=1$$

$$C(n,k)=C(n-1, k-1)+C(n-1, k)$$

```
long C(int n, int k) {  
    if ((k==0) || (k==n))  
        return 1;  
    else  
        return C(n - 1, k - 1) + C(n - 1, k);  
}
```

Coefficienti binomiali – albero ricorsione per $C(5,2)$



Coefficienti binomiali – analisi

$$T(n, k) = C(n, k) = \frac{n!}{k!(n-k)!}$$

Caso peggiore con $k=n/2$ (n pari)

$$T(n, n/2) = \frac{n!}{(n/2)!(n/2)!} \rightarrow \Theta\left(\frac{2^n}{\sqrt{n}}\right) \geq 2^{\frac{n}{2}}$$

Coefficienti binomiali – memoization

```
ResultEntry {  
    boolean done;  
    long value;  
}
```

```
ResultEntry[n+1][k+1] result;
```

Risultati sotto-problemi in una tabella:

$\text{result}[i][j]$ rappresenta $C(i,j)$

Tutte le posizioni della tabella sono inizializzate
con $\text{result}[i][j].done = \text{false}$

- 2 varianti:
- ricorsive + Tabelle
 - iterative + Tabelle

oppure
 $\text{value} = -1$

ResultEntry
{ done : boolean
value : long }

Coefficienti binomiali – memoization (sempre ricorsivo)

```
long C(int n, int k) {  
    if (valore già in ResultEntry????)  
       ????  
  
    if (valori n, k per cui C(n,k) si sa fare????) {  
       ????  
  
        per gli altri casi?  
    }  
}
```

$$\begin{aligned}C(n,k) &= C(n-1, k-1) + C(n-1, k) \\C(n,0) &= 1 \\C(n,n) &= 1\end{aligned}$$

Coefficienti binomiali – memoization (sempre ricorsivo)

```
long C(int n, int k) {  
    if (result[n][k].done == true)  
        return result[n][k].value;  
  
    if (valori n, k per cui C(n,k) si sa fare????) {  
        ????  
  
        per gli altri casi?  
    }  
}
```

$$\begin{aligned}C(n,k) &= C(n-1, k-1) + C(n-1, k) \\C(n,0) &= 1 \\C(n,n) &= 1\end{aligned}$$

Non ricalcola, ma
accede valori in tabella

Coefficienti binomiali – memoization (sempre ricorsivo)

```
long C(int n, int k) {  
  
    if (result[n][k].done == true)  
        return result[n][k].value;  
  
    if ((k == 0) || (k == n)) {  
        result[n][k].done = true;  
        result[n][k].value = 1;  
        return result[n][k].value; }  
  
    per gli altri casi?  
}
```

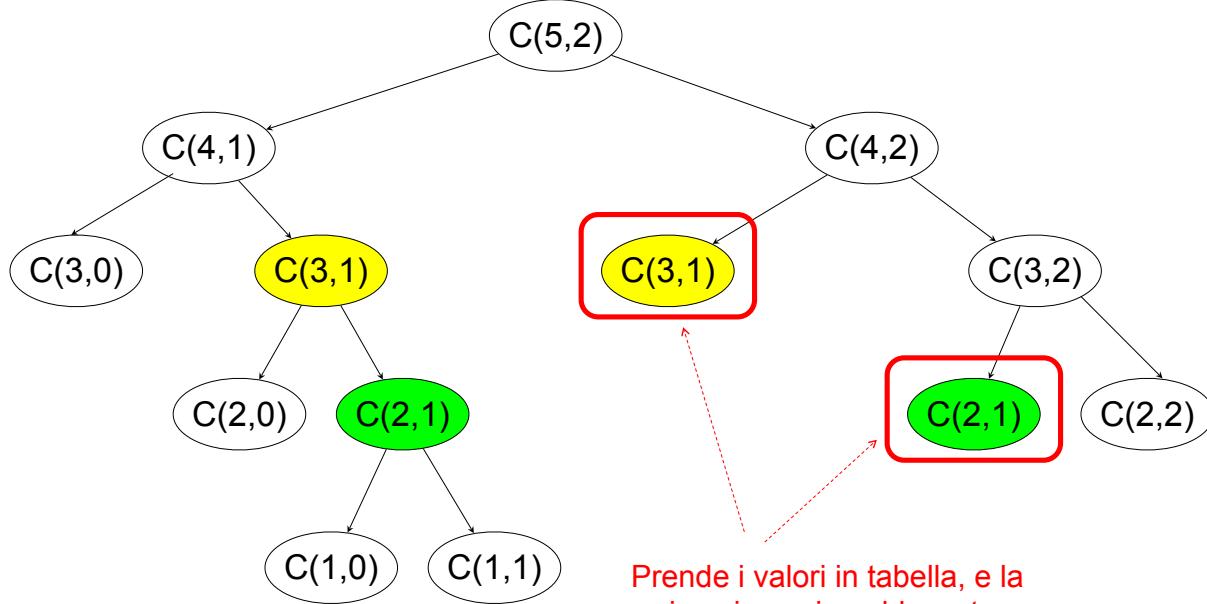
$$\begin{aligned}C(n,k) &= C(n-1, k-1) + C(n-1, k) \\C(n,0) &= 1 \\C(n,n) &= 1\end{aligned}$$


Coefficienti binomiali – memoization (sempre ricorsivo)

```
long C(int n, int k) {  
  
    if (result[n][k].done == true)  
        return result[n][k].value;  
  
    if ((k == 0) || (k == n)) {  
        result[n][k].done = true;  
        result[n][k].value = 1;  
        return result[n][k].value; }  
  
    result[n][k].done = true;  
    result[n][k].value = C(n - 1, k - 1) + C(n - 1, k);  
    return result[n][k].value;  
}
```

$$\begin{aligned}C(n,k) &= C(n-1, k-1) + C(n-1, k) \\C(n,0) &= 1 \\C(n,n) &= 1\end{aligned}$$

Coefficienti binomiali – albero ricorsione con memoization



Programmazione Dinamica

- Eliminiamo ora la ricorsione con **iterazione**
- Bisogna capire come inserire i valori in tabella in modo che la tabella sia utile
 - **Programmazione Dinamica** → ordine di visita
spesso non avrò

$$C(u, k)$$

$$k \leq u$$

Coefficienti binomiali – top-down, riempire la tabella

- `result[i][j]` memorizza il valore di $C(i,j)$
- La tabella ha $n+1$ righe and $k+1$ colonne, $k \leq n$
- **Inizializzazione:** $C(i,0)=1$ e $C(i,i)=1$, per $1 \leq i \leq n$

	0	1		k	n
0	1				
1	1	1			
i	1	1	1		
n	1	1	1	1	

Valori da calcolare

$$C(n,k)=C(n-1, k-1)+C(n-1, k)$$

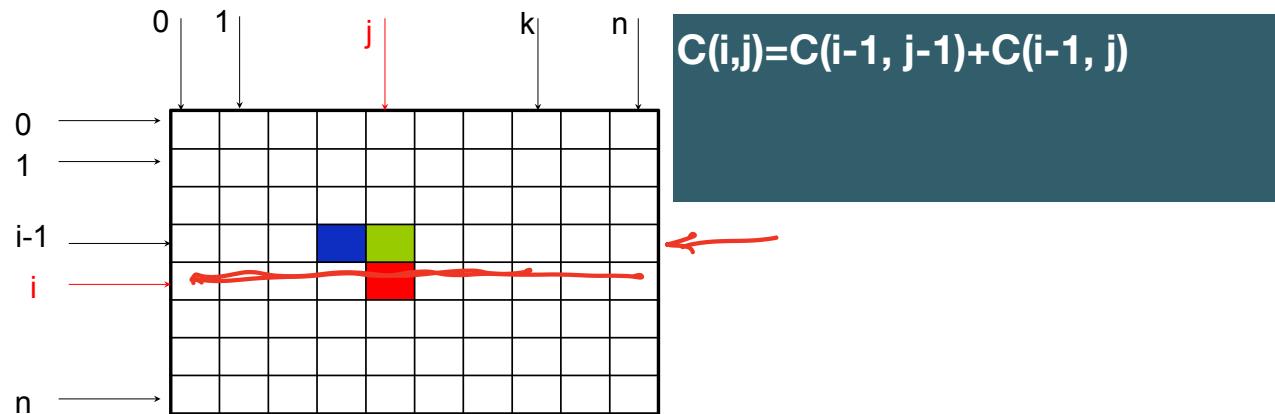
$$C(n,0)=1$$

$$C(n,n)=1$$

Triangolare inferiore
 $k \leq n$

Coefficienti binomiali – top-down, riempire la tabella

- `result[i][j]` memorizza il valore di $C(i,j)$
- Il resto dei valori (i,j) , $2 \leq i \leq n$ e $1 \leq j \leq i-1$ sono calcolati utilizzando i valori $(i-1, j-1)$ e $(i-1, j)$



Coefficienti binomiali – top-down

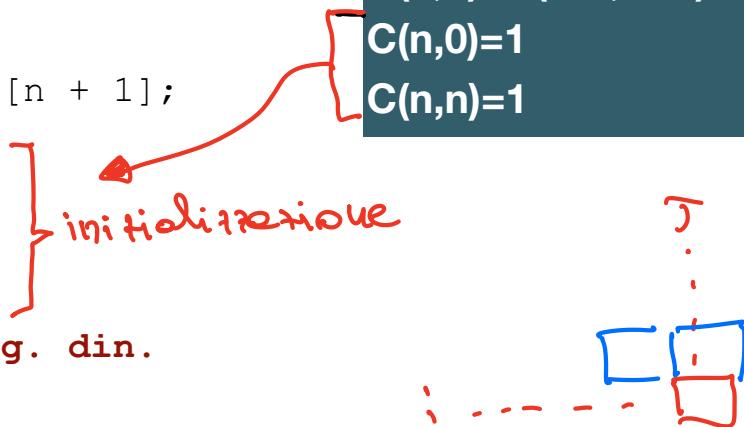
```
long[][] result;  
  
long C(int n, int k) {  
  
    result = new long [n + 1] [n + 1];  
    int i, j;  
    inizializzazione  
  
  
  
  
  
  
riempimento tabella prog. din.  
  
  
  
  
  
  
return result[n] [k];  
}
```

$$\begin{aligned}C(i,j) &= C(i-1, j-1) + C(i-1, j) \\C(n,k) &= C(n-1, k-1) + C(n-1, k) \\C(n,0) &= 1 \\C(n,n) &= 1\end{aligned}$$

Coefficienti binomiali – top-down

```
long[][] result;  
  
long C(int n, int k) {  
  
    result = new long [n + 1] [n + 1];  
    int i, j;  
    for (i=0; i<=n; i++) {  
        result[i][0]=1;  
        result[i][i]=1;  
    }  
    riempimento tabella prog. din.  
  
    return result[n][k];  
}
```

$$\begin{aligned}C(i,j) &= C(i-1, j-1) + C(i-1, j) \\C(n,k) &= C(n-1, k-1) + C(n-1, k) \\C(n,0) &= 1 \\C(n,n) &= 1\end{aligned}$$



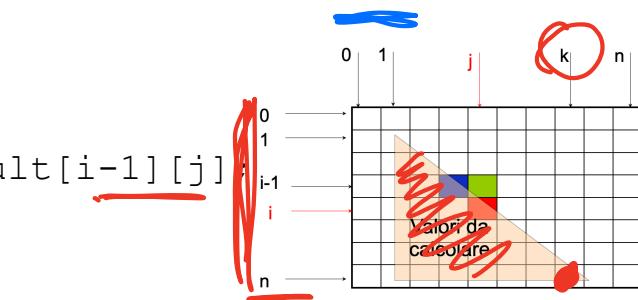
Coefficienti binomiali – top-down

```
long[][] result;  
  
long C(int n, int k) {  
  
    result = new long [n + 1] [n + 1];  
    int i, j;  
    for (i=0; i<=n; i++) {  
        result[i][0]=1;  
        result[i][i]=1;  
    }  
    for (i=2; i<=n; i++)  
        for (j=1; j<i; j++)  
            result[i][j]=result[i-1][j-1]+result[i-1][j];  
  
    return result[n][k];  
}
```

$$\begin{aligned}C(i,j) &= C(i-1, j-1) + C(i-1, j) \\C(n,k) &= C(n-1, k-1) + C(n-1, k) \\C(n,0) &= 1 \\C(n,n) &= 1\end{aligned}$$

Tempo: $O(n^2)$ (o $O(n^k)$)

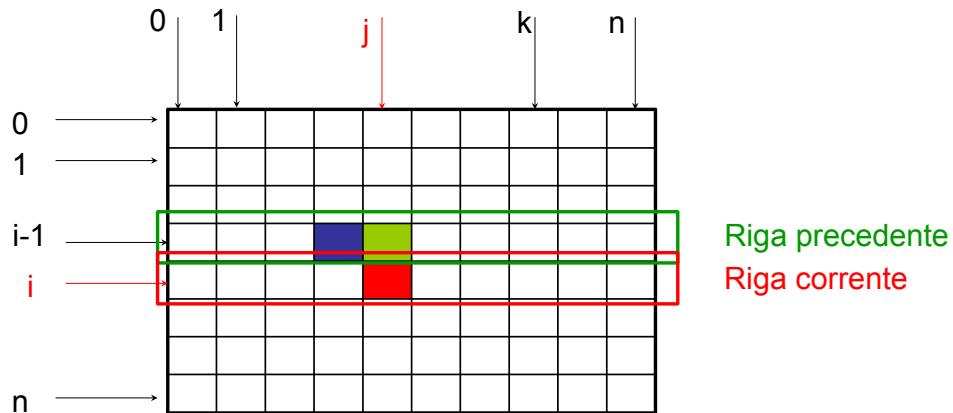
Spazio: $O(n^2)$ (o $O(n^k)$)



Coefficienti binomiali – spazio

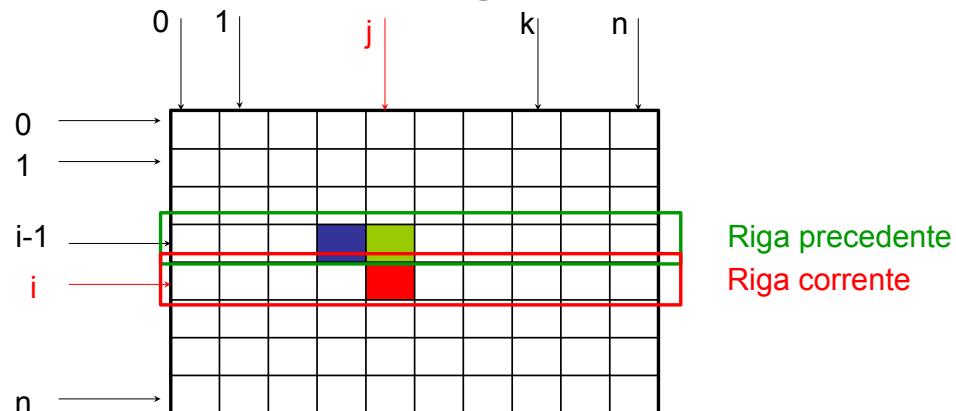
- Di quanti valori c'è effettivamente bisogno per calcolare ogni (i,j) ?
- Abbiamo bisogno di tutta la tabella in ogni momento?

```
result[i][j]=result[i-1][j-1]+result[i-1][j];
```



Coefficienti binomiali – spazio

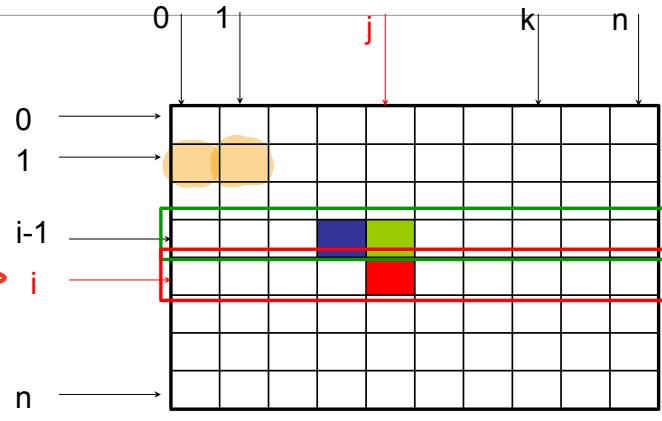
- Ad ogni iterazione `for i`, per calcolare i valori della riga corrente abbiamo bisogno solo dei valori della riga precedente
- Quindi, 2 righe sono sufficienti come buffer di memoria
- Questi buffer sono **riutilizzati** ad ogni iterazione



Coefficienti binomiali – top-down, spazio efficiente

```
long C(int n, int k) {  
  
    long[] result1 = new long[n + 1];  
    long[] result2 = new long[n + 1];  
    result1[0] = 1;  
    result1[1] = 1;  
  
    for (int i = 2; i <= n; i++) {  
        result2[0] = 1; prima colonna  
        for (int j = 1; j < i; j++)  
            result2[j] = result1[j - 1] + result1[j];  
        result2[i] = 1; diagonale  
        long[] auxi = result1;  
        result1 = result2;  
        result2 = auxi;  
    }  
    return result1[k];  
}
```

result1 → i-1
result2 → i



Time $O(n^2)$ o $O(n \cdot k)$
Spazio $O(n)$ o $O(k)$

Programmazione Dinamica in 4 passi

1. **Caratterizzare** la struttura di una soluzione ottima
2. **Ricorsivamente** definire il valore di una soluzione ottima
3. **Calcolare** il valore di una soluzione ottima in modo bottom-up
4. **Costruire** gli elementi della soluzione ottima dalle informazioni calcolate
(non sempre richiesto)

Concatenazione del prodotto di matrici *(a cosa)*

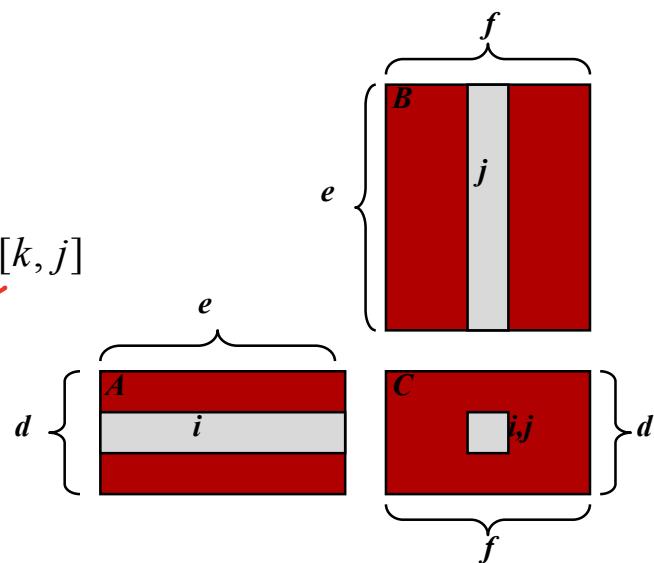
Sequenza ottima di moltiplicazioni di matrici

- Moltiplicazione di matrici

- $C = A * B$
- $A: d \times e$ e $B: e \times f$
- $O(d \cdot e \cdot f)$ tempo

$$C[i, j] = \sum_{k=0}^{e-1} A[i, k] * B[k, j]$$

$\#a$ $\#b$ $\#c$



Sequenza ottima di moltiplicazioni di matrici

Vogliamo calcolare il prodotto di n matrici A_i di dimensione $d_i \times d_{i+1}$

$$A^* = A_0 \times A_1 \times \cdots \times A_{n-1}$$

Hp. semplificativa: il costo di moltiplicazione di due matrici di dimensione $r \times s$ e $s \times t$ è pari a $r \times s \times t$

Sequenza ottima di moltiplicazioni di matrici

- **Sequenza di moltiplicazione di matrici:**
 - Calcolare $A = A_0 * A_1 * \dots * A_{n-1}$
 - A_i è $d_i \times d_{i+1}$
 - Problema: come moltiplicare efficientemente?
- Esempio
 - B è 3×100
 - C è 100×5 B x C ha dimensione 3×5
 - D è 5×5 C x D ha dimensione 100×5
 - $(B * C) * D$ abbiamo $1500 + 75 = 1575$ operazioni
 - $B * (C * D)$ abbiamo $1500 + 2500 = 4000$ operazioni

hp costo: **r x s x t**

Sequenza ottima di moltiplicazioni di matrici

Vogliamo calcolare il prodotto di n matrici A_i di dimensione $d_i \times d_{i+1}$

$$A^* = A_0 \times A_1 \times \cdots \times A_{n-1}$$

Hp. semplificativa: *il costo di moltiplicazione di due matrici di dimensione $r \times s$ e $s \times t$ è pari a $r \times s \times t$*

Esempio: $d_0=100, d_1=20, d_2=1000, d_3=2, d_4=50$

$$(A_0 \times (A_1 \times (A_2 \times A_3)))$$

$$d_2 d_3 d_4 + d_1 d_2 d_4 + d_0 d_1 d_4 = 1.200.000$$

$$(A_0 \times ((A_1 \times A_2) \times A_3))$$

$$d_1 d_2 d_3 + d_1 d_3 d_4 + d_0 d_1 d_4 = 142.000$$

$$((A_0 \times A_1) \times (A_2 \times A_3))$$

$$d_0 d_1 d_2 + d_2 d_3 d_4 + d_0 d_2 d_4 = 7.100.000$$

$$(((A_0 \times A_1) \times A_2) \times A_3)$$

$$d_0 d_1 d_2 + d_0 d_2 d_3 + d_0 d_3 d_4 = 2.210.000$$

$$((A_0 \times (A_1 \times A_2)) \times A_3)$$

$$d_1 d_2 d_3 + d_0 d_1 d_3 + d_0 d_3 d_4 = 54.000$$

Approccio enumerativo

- **Algoritmo:**
 - Prova tutti i possibili modi per moltiplicare $A=A_0 * A_1 * \dots * A_{n-1}$
 - Calcola il numero di operazioni per ogni possibile modo
 - Scegli il migliore
- Costo:
 - Il numero di possibili modi è uguale al numero di alberi binari con n nodi
 - Questo è **esponenziale!**
 - Si chiama numero di Catalan, ed è circa 4^n
 - **Pessimo** algoritmo!

Sequenza ottima di moltiplicazioni

- Il **costo varia** a seconda di come **associamo** il prodotto utilizzando la moltiplicazione tra due matrici: in generale, qual è il **costo minimo**?
- **Nuovo problema:**
 - **input:** sequenza di interi positivi d_0, d_1, \dots, d_n (dove $d_i \times d_{i+1}$ è la taglia della matrice A_i)
 - **output:** minimo numero di operazioni per calcolare

$$A^* = A_0 \times A_1 \times \cdots \times A_{n-1}$$

- (detto **costo minimo** per A^*) con l'ipotesi che la moltiplicazione di due matrici di taglia rxs e sxt richieda $r \times s \times t$ operazioni
- **Nota:** il tutto si applica anche con **algoritmi di moltiplicazione di matrici più veloci**

1. Caratterizzare i sotto-problemi ottimi

- Metodo più **efficiente**: per $0 \leq i \leq j \leq n-1$, sia

$$M(i,j) = \text{costo minimo per } A_i \times A_{i+1} \times \dots \times A_j$$

(la cui dimensione risultante è $d_i \times d_{j+1}$)

- Il **costo minimo** per A^* è quindi dato da **$M(0,n-1)$**

1. Caratterizzare i sotto-problemi ottimi

- Quali sono i sotto-problemi “banali”? Per quali valori di i e j ????

1. Caratterizzare i sotto-problemi ottimi

- **M(i,i) = 0 per 0 ≤ i ≤ n-1** (per ottenere A_i non dobbiamo moltiplicare)

1. Caratterizzare i sotto-problemi ottimi

- $M(i,i) = 0$ per $0 \leq i \leq n-1$ (per ottenere A_i non dobbiamo moltiplicare)
- Caso generale: come possiamo definire i sotto-problemi?

Come definire **ricorsivamente** $A_i \times A_{i+1} \times \dots \times A_j$
in termini di **problemi più piccoli**?

1. Caratterizzare i sotto-problemi ottimi

- $M(i,i) = 0$ per $0 \leq i \leq n-1$ (per ottenere A_i non dobbiamo moltiplicare)
- Caso generale: per ogni $i \leq r < j$, vale

$$\underbrace{A_i \times A_{i+1} \times \dots \times A_j}_{\text{dim: ????}} = \quad \quad \quad \times \quad \quad \quad \underbrace{\quad \quad \quad}_{\text{dim: ????}}$$

↑

costo: ?????

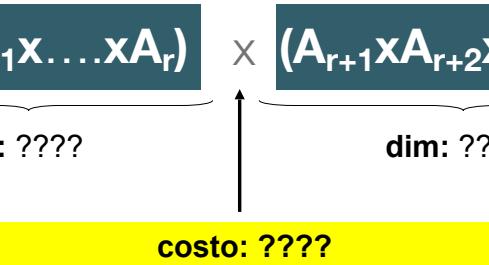
Come possiamo dividere $A_i \times A_{i+1} \times \dots \times A_j$?????

2. Definire la struttura ricorsiva

- $M(i,i) = 0$ per $0 \leq i \leq n-1$ (per ottenere A_i non dobbiamo moltiplicare)
- Caso generale: per ogni $i \leq r < j$, vale

$$\underbrace{A_i x A_{i+1} x \dots x A_j}_{\text{dim: ????}} = (\underbrace{A_i x A_{i+1} x \dots x A_r}_{\text{dim: ????}}) \times (\underbrace{A_{r+1} x A_{r+2} x \dots x A_j}_{\text{dim: ????}})$$

costo: ????



2. Definire la struttura ricorsiva

- $M(i,i) = 0$ per $0 \leq i \leq n-1$ (per ottenere A_i non dobbiamo moltiplicare)
- Caso generale: per ogni $i \leq r < j$, vale

$$\underbrace{A_i x A_{i+1} x \dots x A_j}_{\text{dim: } d_i \times d_{j+1}} = (\underbrace{A_i x A_{i+1} x \dots x A_r}_{\text{dim: ?????}}) \times (\underbrace{A_{r+1} x A_{r+2} x \dots x A_j}_{\text{dim: ?????}})$$

costo: ?????

2. Definire la struttura ricorsiva

- $M(i,i) = 0$ per $0 \leq i \leq n-1$ (per ottenere A_i non dobbiamo moltiplicare)
- Caso generale: per ogni $i \leq r < j$, vale

$$\underbrace{A_i \times A_{i+1} \times \dots \times A_j}_{\text{dim: } d_i \times d_{j+1}} = \underbrace{(A_i \times A_{i+1} \times \dots \times A_r)}_{\text{dim: } d_i \times d_{r+1}} \times \underbrace{(A_{r+1} \times A_{r+2} \times \dots \times A_j)}_{\text{dim: } ???}$$

↑
costo: ???

2. Definire la struttura ricorsiva

- $M(i,i) = 0$ per $0 \leq i \leq n-1$ (per ottenere A_i non dobbiamo moltiplicare)
- Caso generale: per ogni $i \leq r < j$, vale

$$\underbrace{A_i \times A_{i+1} \times \dots \times A_j}_{\text{dim: } d_i \times d_{j+1}} = (\underbrace{A_i \times A_{i+1} \times \dots \times A_r}_{\text{dim: } d_i \times d_{r+1}}) \times (\underbrace{A_{r+1} \times A_{r+2} \times \dots \times A_j}_{\text{dim: } d_{r+1} \times d_{j+1}})$$

costo: ????

2. Definire la struttura ricorsiva

- $M(i,i) = 0$ per $0 \leq i \leq n-1$ (per ottenere A_i non dobbiamo moltiplicare)
- Caso generale: per ogni $i \leq r < j$, vale

$$\underbrace{A_i \times A_{i+1} \times \dots \times A_j}_{\text{dim: } d_i \times d_{j+1}} = (\underbrace{A_i \times A_{i+1} \times \dots \times A_r}_{\text{dim: } d_i \times d_{r+1}}) \times (\underbrace{A_{r+1} \times A_{r+2} \times \dots \times A_j}_{\text{dim: } d_{r+1} \times d_{j+1}})$$

costo: $d_i \times d_{r+1} \times d_{j+1}$

$$M(i,j) = \quad \quad \quad$$

2. Definire la struttura ricorsiva

- $M(i,i) = 0$ per $0 \leq i \leq n-1$ (per ottenere A_i non dobbiamo moltiplicare)
- Caso generale: per ogni $i \leq r < j$, vale

$$\underbrace{A_i \times A_{i+1} \times \dots \times A_j}_{\text{dim: } d_i \times d_{j+1}} = (\underbrace{A_i \times A_{i+1} \times \dots \times A_r}_{\text{dim: } d_i \times d_{r+1}}) \times (\underbrace{A_{r+1} \times A_{r+2} \times \dots \times A_j}_{\text{dim: } d_{r+1} \times d_{j+1}})$$

costo: $d_i \times d_{r+1} \times d_{j+1}$

$$M(i,j) = M(i,r) + M(r+1,j) + d_i d_{r+1} d_{j+1}$$

Per ottenere il **minimo** per $M(i,j)$, bisogna trovare r che lo minimizza

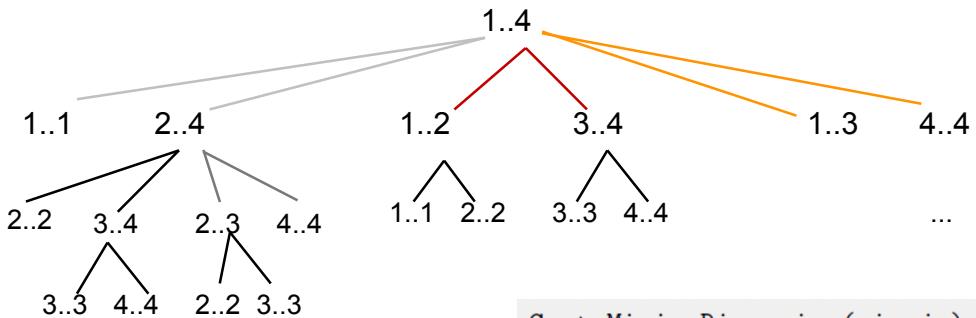
2. Definire la struttura ricorsiva

```
1 CostoMinimoRicorsivo( i, j ):                                ⟨pre: 
2     IF (i >= j) RETURN 0;
3     minimo = +∞;
4     FOR (r = i; r < j; r = r+1) {
5         costo = CostoMinimoRicorsivo( i, r );
6         costo = costo + CostoMinimoRicorsivo( r+1, j );
7         costo = costo + d[i] × d[r+1] × d[j+1];
8         IF (costo < minimo) minimo = costo;
9     }
10    RETURN minimo;
```

$$M(i,j) = M(i,r) + M(r+1,j) + d_i d_{r+1} d_{j+1}$$

Per ottenere il **minimo** per $M(i,j)$, bisogna trovare r che lo minimizza

2. Definire la struttura ricorsiva – top-down (overlap delle soluzioni)



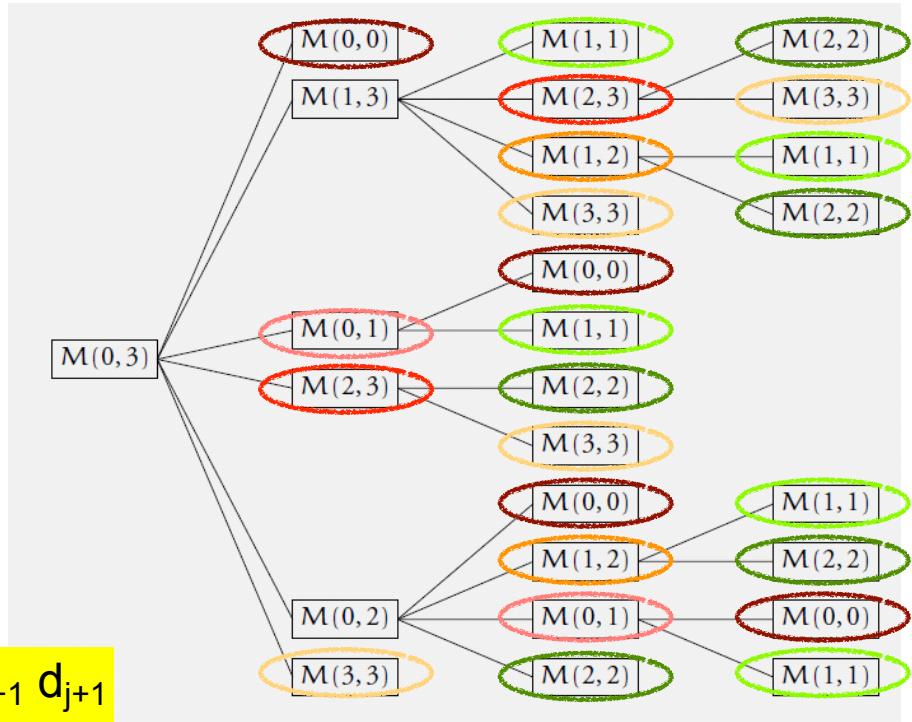
```
CostoMinimoRicorsivo( i, j ):  
    IF ( i >= j ) RETURN 0;  
    minimo = +∞;  
    FOR ( r = i; r < j; r = r+1 ) {  
        costo = CostoMinimoRicorsivo( i, r );  
        costo = costo + CostoMinimoRicorsivo( r+1, j );  
        costo = costo + d[i] × d[r+1] × d[j+1];  
        IF (costo < minimo) minimo = costo;  
    }  
    RETURN minimo;
```

2. Definire la struttura ricorsiva — top-down (overlap delle soluzioni)

Numero esponenziale di chiamate ricorsive, ciascuna invocata molte volte con gli stessi parametri d'ingresso

```
CostoMinimoRicorsivo( i, j ):  
    IF (i >= j) RETURN 0;  
    minimo = +∞;  
    FOR (r = i; r < j; r = r+1) {  
        costo = CostoMinimoRicorsivo( i, r );  
        costo = costo + CostoMinimoRicorsivo( r+1, j );  
        costo = costo + d[i] × d[r+1] × d[j+1];  
        IF (costo < minimo) minimo = costo;  
    }  
    RETURN minimo;
```

$$M(i,j) = M(i,r) + M(r+1,j) + d_i d_{r+1} d_{j+1}$$



2. Definire la struttura ricorsiva — top-down (overlap delle soluzioni)

$$M(i,j) = M(i,r) + M(r+1,j) + d_i d_{r+1} d_{j+1}$$

- Quindi, indicando con $T(n)$ la complessità del calcolo di $M(0,n-1)$,
 - Qual è l'equazione di ricorrenza? (*ricordate che $M(i,j)$ prova tutti gli r*)

$$T(n) = c' + \text{????} =$$



Inizializzazioni

2. Definire la struttura ricorsiva — top-down (overlap delle soluzioni)

$$M(i,j) = M(i,r) + M(r+1,j) + d_i d_{r+1} d_{j+1}$$

- Quindi, indicando con $T(n)$ la complessità del calcolo di $M(0,n-1)$,
 - Qual è l'equazione di ricorrenza? (*ricordate che $M(i,j)$ prova tutti gli r*)

$$T(n) = c' + \sum_{r=0,..,n-2} (T(r+1) + T(n-r-1) + c'')$$

- Esponenziale!!

3. Calcolare valore ottimo della soluzione — bottom-up

- Il problema, come visto, è che i valori vengono calcolati più volte
- Ricorriamo alla programmazione *dinamica*
- Memorizziamo i risultati parziali in una matrice
 - Il calcolo di $M[i,j]$ si basa su valori di M già calcolati
 - Cerchiamo di capire quali sono i valori *base*
- **Domanda:** In questo caso, quali elementi della matrice possiamo riempire subito?

3. Calcolare valore ottimo della soluzione — bottom-up

- Il problema, come visto, è che i valori vengono calcolati più volte
- Ricorriamo alla programmazione *dinamica*
- Memorizziamo i risultati parziali in una matrice
 - Il calcolo di $M[i,j]$ si basa su valori di M già calcolati
 - Cerchiamo di capire quali sono i valori *base*
 - **$M[i,i]=0$ (era il caso base della ricorsione)**
 - Calcolare il pro...
sola matrice ha

Quindi, al passo 0, possiamo riempire
con 0 la diagonale della matrice M

3. Calcolare valore ottimo della soluzione — bottom-up

tabella: costi[i][j] = M(i,j)

i - j	0	1	2	3
0	0			
1		0		
2			0	
3				0

$$M(i,j) = M(i,r) + M(r+1,j) + d_i \ d_{r+1} \ d_{j+1}, \ i < j$$

3. Calcolare valore ottimo della soluzione — bottom-up

tabella: costi[i][j] = M(i,j)

Domanda: noti i valori della diagonale (tutti 0), quali elementi della matrice possiamo calcolare?

In altre parole, quali elementi della matrice riempire al passo 1?

M[0,3] di quali valori ha bisogno?

NON possibile
al passo 1!!!!

i - j	0	1	2	3
0	0			
1		0		
2			0	
3				0

$$M(i,j) = M(i,r) + M(r+1,j) + d_i \ d_{r+1} \ d_{j+1}, \ i < j$$

3. Calcolare valore ottimo della soluzione — bottom-up

tabella: costi[i][j] = M(i,j)

M[0,3] di quali
valori ha bisogno?

NON possibile
al passo 1!!!!

Io devo calcolare per ogni $i \leq r < j$

$$M(0,3) = M(0,0) + M(1,3), r=0$$

$$M(0,3) = M(0,1) + M(2,3), r=1$$

$$M(0,3) = M(0,2) + M(3,3), r=2$$

$i - j$	0	1	2	3
0				
1				
2				
3				

A diagram showing a 5x5 grid of colored cells. The columns are labeled i-j (0, 1, 2, 3) and the rows are labeled i-j (0, 1, 2, 3). The cells are colored in a checkerboard pattern: (0,0) is dark grey, (0,1) is light grey, (0,2) is dark grey, (0,3) is light grey; (1,0) is light grey, (1,1) is dark grey, (1,2) is light grey, (1,3) is dark grey; (2,0) is dark grey, (2,1) is light grey, (2,2) is dark grey, (2,3) is light grey; (3,0) is light grey, (3,1) is dark grey, (3,2) is light grey, (3,3) is dark grey. A diagonal line of zeros starts from the cell (1,1) and extends to the cell (3,3).

$$M(i,j) = M(i,r) + M(r+1,j) + d_i d_{r+1} d_{j+1}, i < j$$

3. Calcolare valore ottimo della soluzione — bottom-up

tabella: $\text{costi}[i][j] = M(i,j)$

$M[2,3]$ di quali valori ha bisogno?

SI!!!!

Io devo calcolare per ogni $i \leq r < j$

$$M(2,3) = M(2,2) + M(3,3), r=2$$

$M[1,2]$ di quali valori ha bisogno?

SI!!!!

Io devo calcolare per ogni $i \leq r < j$

$$M(1,2) = M(1,1) + M(2,2), r=2$$

$i - j$	0	1	2	3
0	0			
1		0		
2			0	
3				0

$$M(i,j) = M(i,r) + M(r+1,j) + d_i d_{r+1} d_{j+1}, i < j$$

3. Calcolare valore ottimo della soluzione — bottom-up

tabella: costi[i][j] = M(i,j)

Domanda: noti i valori della diagonale (tutti 0), quali elementi della matrice possiamo calcolare?

In altre parole, quali elementi della matrice riempire al passo 1?

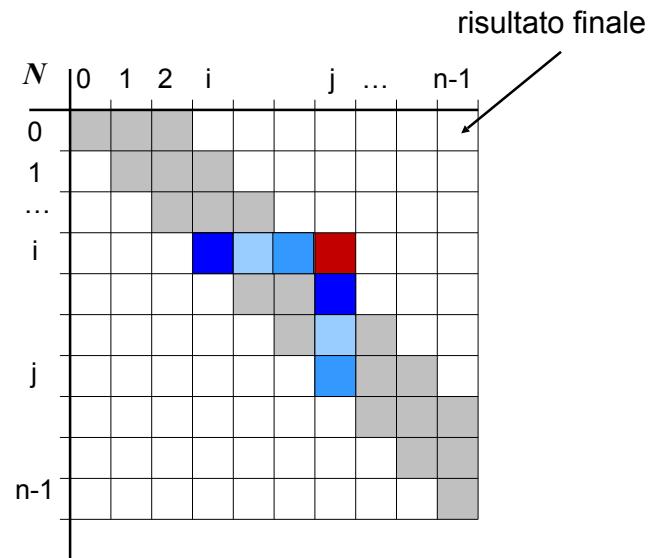
Per diagonali!!!!

i — j	0	1	2	3
0	0			
1		0		
2			0	
3				0

$$M(i,j) = M(i,r) + M(r+1,j) + d_i \ d_{r+1} \ d_{j+1}, \ i < j$$

3. Calcolare valore ottimo della soluzione — bottom-up

- L'approccio bottom-up riempie la matrice per diagonali
- Valore $[i,j]$ calcolato con i valori precedenti in riga i -esima e colonna j -esima



Soluzione

```
1 CostoMinimoIterativo( ):  
2     FOR (i = 0; i < n; i = i+1) {  
3         costi[i][i] = 0;  
4     }  
5     FOR (diagonale = 1; diagonale < n; diagonale = diagonale+1) {  
6         FOR (i = 0; i < n-diagonale; i = i+1) {  
7             j = i + diagonale;  
8             costi[i][j] = +∞;  
9             FOR (r = i; r < j; r = r+1) {  
10                 costo = costi[i][r] + costi[r+1][j]; ————— Sostituiscono le  
11                 costo = costo + d[i] × d[r+1] × d[j+1]; ————— chiamate ricorsive  
12                 IF (costo < costi[i][j]) {  
13                     costi[i][j] = costo;  
14                     indice[i][j] = r; —————— Serve a ricostruire la  
15                 }  
16             }  
17         }  
18     }  
19     RETURN costi[0][n-1];
```

i - j	0	1	2	3
0	0			
1		0		
2			0	
3				0

Analisi di complessità

- La tabella **costi** ha dimensione $n \times n$ e il calcolo di ciascuno dei suoi elementi richiede $O(n)$ tempo (ciclo **for** più interno)
- **CostoMinimoIterativo** richiede $O(n^3)$ tempo (laddove la versione ricorsiva è esponenziale in n)
- Lo **spazio occupato** è $O(n^2)$ celle
- Anche qui ottimizzazione in spazio possibile

```
1 CostoMinimoIterativo( ):
2     FOR (i = 0; i < n; i = i+1) {
3         costi[i][i] = 0;
4     }
5     FOR (diagonale = 1; diagonale < n; diagonale = diagonale+1) {
6         FOR (i = 0; i < n-diagonale; i = i+1) {
7             j = i + diagonale;
8             costi[i][j] = +∞;
9             FOR (r = i; r < j; r = r+1) {
10                 costo = costi[i][r] + costi[r+1][j];
11                 costo = costo + d[i] × d[r+1] × d[j+1];
12                 IF (costo < costi[i][j]) {
13                     costi[i][j] = costo;
14                     indice[i][j] = r;
15                 }
16             }
17         }
18     }
19     RETURN costi[0][n-1];
```

4. Costruire la soluzione ottima

- Utilizzando la tabella **indice**, si ricava la sequenza ottima di moltiplicazioni per il calcolo di A^*
 - Scrivere una procedura **StampaSequenza(indice, i, j)** che restituisca la sequenza di prodotti di costo minimo per moltiplicare le matrici in A_i, \dots, A_j

```
1 CostoMinimoIterativo( ):  
2     FOR (i = 0; i < n; i = i+1) {  
3         costi[i][i] = 0;  
4     }  
5     FOR (diagonale = 1; diagonale < n; diagonale = diagonale+1) {  
6         FOR (i = 0; i < n-diagonale; i = i+1) {  
7             j = i + diagonale;  
8             costi[i][j] = +∞;  
9             FOR (r = i; r < j; r = r+1) {  
10                 costo = costi[i][r] + costi[r+1][j];  
11                 costo = costo + d[i] × d[r+1] × d[j+1];  
12                 IF (costo < costi[i][j]) {  
13                     costi[i][j] = costo;  
14                     indice[i][j] = r;  
15                 }  
16             }  
17         }  
18     }  
19     RETURN costi[0][n-1];
```

4. Costruire la soluzione ottima

```
StampaSequenza(indice,i,j)
IF (i==j) {
    STAMPA Ai;
}
else {
    STAMPA "("
    r=indice[i][j];
    STAMPA StampaSequenza(indice,i,r);
    STAMPA StampaSequenza(indice,r+1,j);
    STAMPA ")"
}
```

Regole auree per la programmazione dinamica

1. Soluzione **ottima** di un problema deriva dalla **ottimalità** delle soluzioni dei suoi **sotto-problemi**: Definizione di una **regola ricorsiva** di calcolo della soluzione ottima (punto 1.)
2. **Tabella di programmazione dinamica** da riempire **iterativamente** con la regola ricorsiva (punto 2.)
3. **Ordine di riempimento** della tabella (punto 3.)
4. Cercare eventualmente la **soluzione di costo ottimo**

Esercizio

$A_0: 30 \times 35$

$A_1: 35 \times 15$

$A_2: 15 \times 5$

$A_3: 5 \times 10$

$A_4: 10 \times 20$

$A_5: 20 \times 25$

Esercizio

	d_0	d_1	d_2	d_3	d_4	d_5	d_6
d_q	0	1	2	3	4	5	0
0	15,750	7,875	9,375	11,875	15,125	0	0
1	0	2,625	4,375	7,125	10,500	"	"
2	0	750	2,500	5,375	0	"	"
3	0	1,000	0	3,500	0	"	"
4	0	0	0	5,000	0	"	"
5	0	0	0	0	0	"	"

$$M[i,r] + M[r+1,j] + d_i \cdot d_{r+1} \cdot d_j$$

Esempio per calcolo $M[1,4]$

$$\begin{aligned} r &= 1 & d_1 \cdot d_2 \cdot d_5 \\ &= 35 \cdot 15 \cdot 20 \\ &= 10500 \end{aligned}$$

$r = 2$

$$\begin{aligned} d_1 \cdot d_3 \cdot d_5 &= \\ &= 35 \cdot 5 \cdot 20 = 3500 \end{aligned}$$

$$\begin{aligned} r &= 3 & d_1 \cdot d_4 \cdot d_5 \\ &= 35 \cdot 10 \cdot 20 = 7000 \end{aligned}$$

Esercizio

$$(A_0^* (A_1^* A_2))^* ((A_3^* A_4)^* A_5)$$

0	1	2	3	4	5	0
0	15,750	7,875	9,375	11,875	15,125	0
1	0	2,625	4,375	7,125	10,500	1
2	0	750	2,500	5,375	0	2
3	0	1,000	0	3,500	0	3
4	0	0	5,000	0	0	4
5	0	0	0	0	0	5

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

Longest Common Subsequence

(2 éléments de contrôles)

Sicurezza dei sistemi e sotto-sequenza comune più lunga (LCS)

- Traccia dei comandi eseguiti (file di *log*) per monitorare eventuali intrusioni (*intrusion detection*)
- Intrusione: **sotto-sequenza** di un file di log

Operazioni rappresentate come etichette: **A**, **B**, ...

Esempio:

log = F =

B, A, A, B, D, C, D, C, A, A, C, A, C, B, A

intrusione = S =

A, D, C, A, A, B

Sotto-sequenza comune più lunga

- Una sequenza S di lunghezza k è **sotto-sequenza** di una sequenza A di lunghezza n se esistono delle posizioni $0 \leq i_0 < i_1 < \dots < i_{k-1} \leq n-1$ in A tali che
$$S[j] = A[i_j] \text{ per } j = 0, 1, \dots, k-1$$
- S è **sotto-sequenza comune** ad A e B, se è sotto-sequenza comune sia di A che di B

$$A = \langle A, B, C, B, D, A, B \rangle \quad A = \langle A, B, C, B, D, A, B \rangle$$

$$B = \langle B, D, C, A, B, A \rangle \quad B = \langle B, D, C, A, B, A \rangle$$

Sotto-sequenza comune più lunga

- Una sequenza S di lunghezza k è **sotto-sequenza** di una sequenza A di lunghezza n se esistono delle posizioni $0 \leq i_0 < i_1 < \dots < i_{k-1} \leq n-1$ in A tali che
$$S[j] = A[i_j] \text{ per } j = 0, 1, \dots, k-1$$
- S è **sotto-sequenza comune** ad A e B, se è sotto-sequenza comune sia di A che di B
- $\text{LCS}(A, B) = \text{lunghezza della sotto-sequenza comune più lunga}$ (*longest common subsequence*)

Sotto-sequenza comune più lunga

- $\text{LCS}(A, B) = \text{lunghezza della sotto-sequenza comune più lunga}$ (*longest common subsequence*)

$$A = \langle A, \mathbf{B}, \mathbf{C}, B, D, \mathbf{A}, B \rangle \quad A' = \langle A, \mathbf{B}, \mathbf{C}, \mathbf{B}, D, \mathbf{A}, B \rangle$$

$$S = \langle B, C, A \rangle$$

$$S' = \langle B, C, B, A \rangle$$

$$B = \langle \mathbf{B}, D, \mathbf{C}, \mathbf{A}, B, A \rangle$$

$$B' = \langle \mathbf{B}, D, \mathbf{C}, A, \mathbf{B}, \mathbf{A} \rangle$$

S è LCS di A e B?



Sotto-sequenza comune più lunga

- $\text{LCS}(A, B) = \text{lunghezza}$ della **sotto-sequenza comune più lunga** (*longest common subsequence*)

$$A = \langle A, \mathbf{B}, \mathbf{C}, B, D, \mathbf{A}, B \rangle \quad A' = \langle A, \mathbf{B}, \mathbf{C}, \mathbf{B}, D, \mathbf{A}, B \rangle$$

$$S = \langle B, C, A \rangle$$

$$S' = \langle B, C, B, A \rangle$$

$$B = \langle \mathbf{B}, D, \mathbf{C}, \mathbf{A}, B, A \rangle$$

$$B' = \langle \mathbf{B}, D, \mathbf{C}, A, \mathbf{B}, \mathbf{A} \rangle$$

S' è LCS di A e B?



Altre applicazioni di LCS

- **Biologia**
 - Sequenze di DNA sono rappresentate come sequenze di sotto-molecole, ognuna appartenente a un tipo: A C G T. In genetica, è di forte interesse calcolare le similarità tra due DNA mediante LCS
- **Confronto tra file**
 - **Sistemi di versionamento: esempio** - "diff" è utilizzato per confrontare due differenti versioni dello stesso file, per determinare quali cambiamenti sono stati fatti al file. Funziona trovando la LCS delle righe dei due file

Soluzione di forza bruta

- Siano A e B due sequenze
 - $\text{LCS}(A,B) = \text{LCS}(m,n)$, con $m = |A|$ e $n = |B|$
- Per ogni sotto-sequenza di A, controlla se è sotto-sequenza di B
 - Ci sono 2^m sotto-sequenze di A da controllare
- Per controllare ogni sotto-sequenza, si impiega $\Theta(n)$ tempo
 - Scandisci B per cercare la prima lettera della sotto-sequenza in esame, e da lì cerca la seconda lettera, e così via
- Tempo: $\Theta(n2^m)$

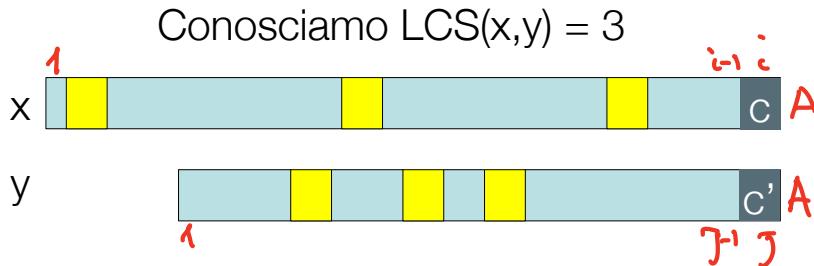
1. e 2. Struttura ricorsiva (e incrementale)

LCS(A,B) può essere ricavato analizzando i prefissi di A e B....

- Immaginiamo di aver trovato l'ottimo tra due sotto-problemi
 - **Prefisso** x di A e **prefisso** y di B
 - $|y| = j - 1, |x| = i - 1$

Analizziamo il carattere successivo ad entrambi i prefissi e consideriamo i due casi $c == c'$ e $c != c'$

Conosciamo $\text{LCS}(x,y) = 3$

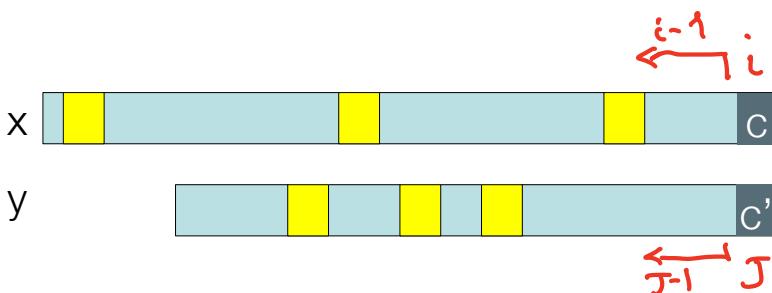


1. e 2. Struttura ricorsiva

LCS(A,B) può essere ricavato analizzando i prefissi di A e B....

- Immaginiamo di aver trovato l'ottimo tra due sotto-problemi
 - **Prefisso** x di A e **prefisso** y di B
 - $|y| = j - 1, |x| = i - 1$

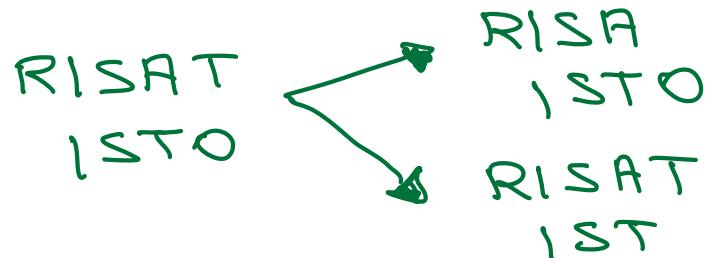
Analizziamo il carattere successivo ad entrambi i prefissi e consideriamo i due casi $c == c'$ e $c != c'$



Se $c == c'$

$$\text{LCS}(i,j) = \text{LCS}(x, y) + 1$$

PER:A
MEL:A

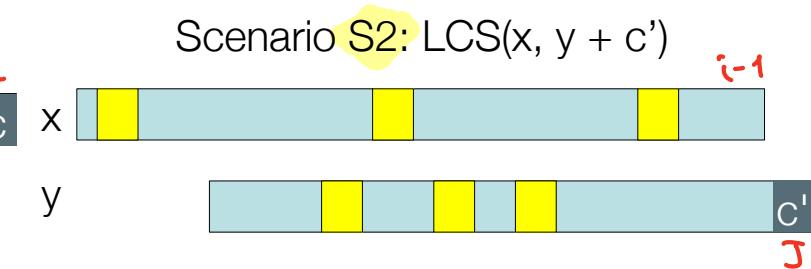
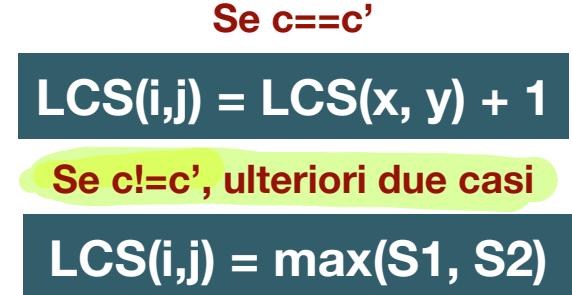
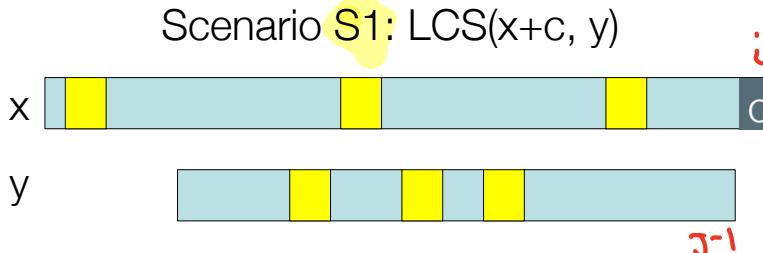


1. e 2. Struttura ricorsiva

LCS(A,B) può essere ricavato analizzando i prefissi di A e B....

- Immaginiamo di aver trovato l'ottimo tra due sotto-problemi
 - **Prefisso** x di A e **prefisso** y di B
 - $|y| = j - 1, |x| = i - 1$

Analizziamo il carattere successivo ad entrambi i prefissi e consideriamo i due casi $c == c'$ e $c != c'$



matrice L , m × n

o se $L[i, j] = \text{LCS Tra}$
 $A[0, i-1]$ e
 $B[0, j-1]$

1. e 2. Struttura ricorsiva

$$\text{LCS}(i, j) = \begin{cases} 0 & \text{se } i=0 \text{ o } j=0 \\ \text{LCS}(i-1, j-1) + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{se } i, j > 0 \text{ e } x_i \neq y_j \end{cases}$$

3. Calcolo bottom-up

Quali sono i casi base (cioè i valori che posso già inserire in L)?

Se una delle due sequenze è vuota....e quindi?

Se $i = 0$ o $j = 0 \rightarrow L(i,j) = 0$

[$A[0,-1] = B[0,-1] = \text{sequenza vuota}$]

	0	1	2		n
0	0	0	0	0	0
1	0				
2	0				
3	0				
4	0				
m	0				

Matrice L di dimensione **mxn**

dove $L(i,j)$ rappresenta $\text{LCS}(A[0, i-1], B[j-1])$

o,

3. Calcolo bottom-up

$$L(i, j) = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ L(i - 1, j - 1) + 1 & \text{se } i, j > 0 \text{ e } a[i - 1] = b[j - 1] \\ \max \{ L(i, j - 1), L(i - 1, j) \} & \text{se } i, j > 0 \text{ e } a[i - 1] \neq b[j - 1] \end{cases}$$

se $i = 0$ o $j = 0$

se $i, j > 0$ e $a[i - 1] = b[j - 1]$

se $i, j > 0$ e $a[i - 1] \neq b[j - 1]$

$B[j-1]$

Tabella di programmazione dinamica di dim
 $(m+1) \times (n+1)$

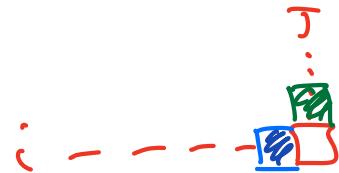
$$L[i][j] = LCS(i, j)$$

Ordine riempimento?

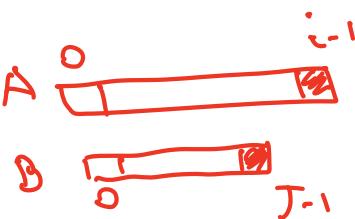
per riga o
colonna

se $m < n$, per riga o per colonne ??

0	1	2	3	4	5	n
0	0	0	0	0	0	0
1	0					
2	0					
3						
4						
5						
m						



3. Calcolo bottom-up



```

1  LCS( a, b ):
2      FOR (i = 0; i <= m; i = i+1)
3          lunghezza[i][0] = 0;
4      FOR (j = 0; j <= n; j = j+1)
5          lunghezza[0][j] = 0;
6      FOR (i = 1; i <= m; i = i+1)
7          FOR (j = 1; j <= n; j = j+1) {
8              IF (a[i-1] == b[j-1]) {
9                  lunghezza[i][j] = lunghezza[i-1][j-1] + 1;
10             } ELSE IF (lunghezza[i][j-1] > lunghezza[i-1][j]) {
11                 lunghezza[i][j] = lunghezza[i][j-1];
12             } ELSE {
13                 lunghezza[i][j] = lunghezza[i-1][j];
14             }
15         }
16     RETURN lunghezza[m][n];
    
```

} per righe \rightarrow col

(pre: a e b sono di lunghezza m e n)

inizializzazione

$$L(i, j) = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ L(i - 1, j - 1) + 1 & \text{se } i, j > 0 \text{ e } a[i - 1] = b[j - 1] \\ \max \{ L(i, j - 1), L(i - 1, j) \} & \text{se } i, j > 0 \text{ e } a[i - 1] \neq b[j - 1] \end{cases}$$

Complessità

- Domanda: (solita)....complessità??

```
1  LCS( a, b ):  
2      FOR (i = 0; i <= m; i = i+1)  
3          lunghezza[i][0] = 0;  
4      FOR (j = 0; j <= n; j = j+1)  
5          lunghezza[0][j] = 0;  
6      FOR (i = 1; i <= m; i = i+1)  
7          FOR (j = 1; j <= n; j = j+1) {  
8              IF (a[i-1] == b[j-1]) {  
9                  lunghezza[i][j] = lunghezza[i-1][j-1] + 1;  
10             } ELSE IF (lunghezza[i][j-1] > lunghezza[i-1][j]) {  
11                 lunghezza[i][j] = lunghezza[i][j-1];  
12             } ELSE {  
13                 lunghezza[i][j] = lunghezza[i-1][j];  
14             }  
15         }  
16     RETURN lunghezza[m][n];
```

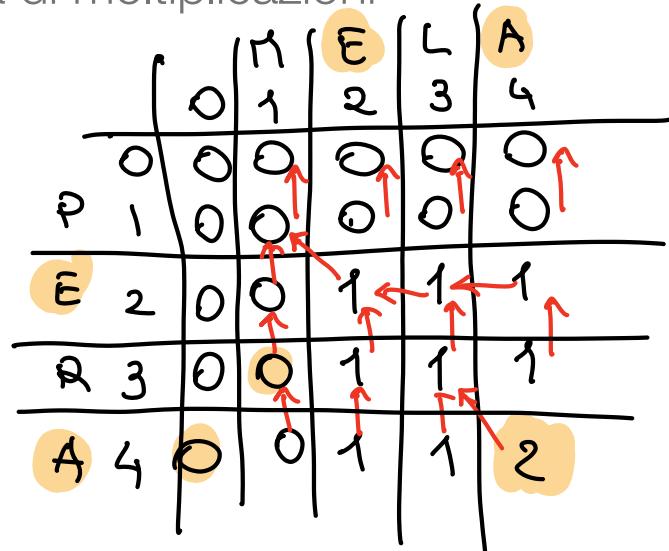
(pre: a e b sono di lunghezza m e n)

- Tempo $O(mn)$
- Spazio $O(mn)$

È possibile ridurre lo spazio a $O(n)$, usando solo le ultime due righe (o colonne)

4. Come ricostruire la sequenza più lunga?

Bisogna utilizzare una matrice **indice** simile a quella utilizzata per la sequenza ottima di moltiplicazioni



Calcolo di indice

```
1  LCS( a, b ):                                ⟨pre: a e b sono di lunghezza m e n⟩
2      FOR (i = 0; i <= m; i = i+1)
3          lunghezza[i][0] = 0;
4      FOR (j = 0; j <= n; j = j+1)
5          lunghezza[0][j] = 0;
6      FOR (i = 1; i <= m; i = i+1)
7          FOR (j = 1; j <= n; j = j+1) {
8              IF (a[i-1] == b[j-1]) {
9                  lunghezza[i][j] = lunghezza[i-1][j-1] + 1;
10             } ELSE IF (lunghezza[i][j-1] > lunghezza[i-1][j]) {
11                 lunghezza[i][j] = lunghezza[i][j-1];
12             } ELSE {
13                 lunghezza[i][j] = lunghezza[i-1][j];
14             }
15         }
16     RETURN lunghezza[m][n];
```

indice[i][j]=<i-1,j-1>

indice[i][j]=<i,j-1>

indice[i][j]=<i-1,j>

Esempio

$$X = \langle A, B, C, B, D, A, B \rangle$$

$$Y = \langle B, D, C, A, B, A \rangle$$

If $x_i = y_j$

indice[i, j] = “\”

Else if $L[i - 1, j] \geq L[i, j-1]$

indice[i, j] = “↑”

else

indice[i, j] = “←”

```

1 LCS( a, b ):                                (pre: a e b sono di lunghezza m e n)
2   FOR ( i = 0; i <= m; i = i+1 )
3     lunghezza[i][0] = 0;
4   FOR ( j = 0; j <= n; j = j+1 )
5     lunghezza[0][j] = 0;
6   FOR ( i = 1; i <= m; i = i+1 )
7     FOR ( j = 1; j <= n; j = j+1 ) {
8       IF ( a[i-1] == b[j-1] ) {
9         lunghezza[i][j] = lunghezza[i-1][j-1] + 1;
10      } ELSE IF ( lunghezza[i][j-1] > lunghezza[i-1][j] ) {
11        lunghezza[i][j] = lunghezza[i][j-1];
12      } ELSE {
13        lunghezza[i][j] = lunghezza[i-1][j];
14      }
15    }
16  RETURN lunghezza[m][n];

```

$$L(i, j) = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ L(i - 1, j - 1) + 1 & \text{se } i, j > 0 \text{ e } a[i - 1] = b[j - 1] \\ \max \{ L(i, j - 1), L(i - 1, j) \} & \text{se } i, j > 0 \text{ e } a[i - 1] \neq b[j - 1] \end{cases}$$

Riempiamo per righe....

	0	1	2	3	4	5	6
0	x _i	0	0	0	0	0	0
1	A	0	0	0	1	-1	1
2	B	0	1	-1	-1	1	2
3	C	0	1	1	2	-2	2
4	B	0	1	1	2	2	-3
5	D	0	1	2	2	2	3
6	A	0	1	2	2	3	4
7	B	0	1	2	2	3	4

Procedura di Stampa

- Inizia da **indice[m, n]** e segui le frecce
- Quando si incontra un “\“ in **indice[i, j]**
 $\Rightarrow x_i = y_j$ è un elemento della LCS

LCS: B C B A

	0	1	2	3	4	5	6
0	x _i	0	0	0	0	0	0
1	y _j	B	D	C	A	B	A
0	0	0	0	0	0	0	0
1	A	0	0	0	1	-1	1
2	B	0	1	-1	-1	1	2
3	C	1	1	2	-2	2	2
4	B	1	1	2	2	3	-3
5	D	1	2	2	2	3	3
6	A	1	2	2	3	3	4
7	B	1	2	2	3	4	4

Procedura di Stampa

1. if $i = 0$ or $j = 0$
2. then return
indice
3. if $b[i, j] = " \backslash "$
4. then Stampa_LCS(indice, A, i - 1, j - 1)
5. print x_i
6. elseif $indice[i, j] = " \uparrow "$
7. then Stampa_LCS(indice, A, i - 1, j)
8. else Stampa_LCS(indice, A, i, j - 1)

Tempo: $\Theta(m+n)$

} perché print
dopo chiamate
ricorsive?

Chiamata iniziale: Stampa_LCS(indice, A, length[X], length[Y])

Provate

. RISOTTO
. RISTORO

		A	B	C	D	A
		0	0	0	0	0
A	0	1	-1	1	1	1
	0	1	2	2	2	2
C	0	1	2	2	2	2
B	0	2	2	2	2	2
D	0	1	2	2	3	3
E	0	1	2	2	3	3
A	0	1	2	2	3	3

LCS- "ACDA"

Edit Distance

Esercizio Programmazione Dinamica

APE vs ARPA



Allineamento tra due sequenze

Misuriamo la similarità

Distanza di edit tra due sequenze S_1 e S_2 :

weighted e. d.

Vogliamo trovare l'allineamento ottimo fra le due sequenze, cioè quello che minimizza la distanza (edit distance) fra di esse:

1. Possiamo inserire spazi nelle sequenze (allo scopo di allinearle)

2. La distanza è la somma delle distanze fra coppie di caratteri allineati:

- **caratteri uguali** sono a distanza 0

- **caratteri diversi** o caratteri allineati a spazio hanno distanza 1

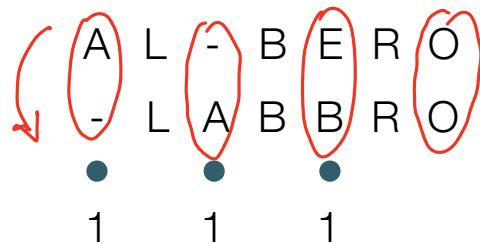
Distanza di edit – ALBERO vs LABBRO

(A)	(L)	B	(E)	R	O
L	A	B	B	R	O
●	●	●			
1	1	1			

- Le stringhe hanno già lunghezza uguale, quindi sono già allineate
- **Costo della distanza?**

la distanza è 3

Distanza di edit – ALBERO vs LABBRO



• In alternativa?

Ricordate la possibilità di usare spazi

la distanza è 3

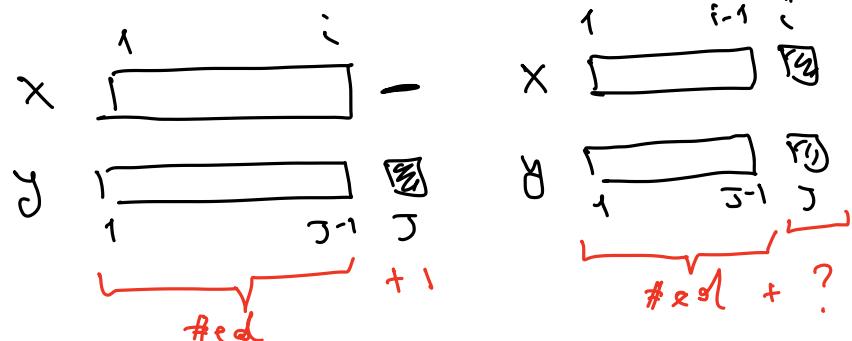
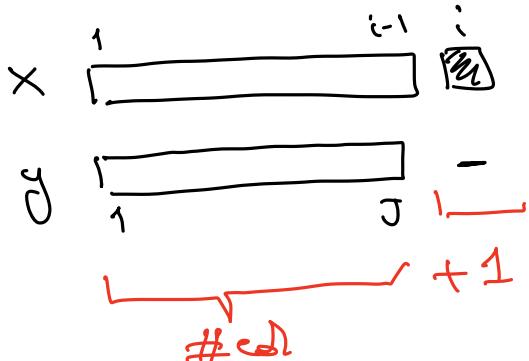
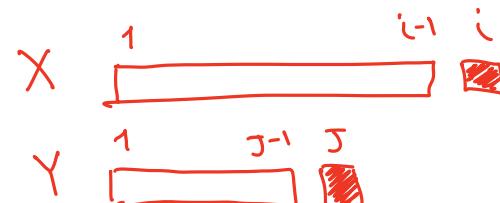
- Altre misure di conto? weighted ED

$$M[i, j] = \min \left\{ \begin{array}{l} 1 + M[i-1, j], \\ 1 + M[i, j-1], \\ p(i, j) + M[i-1, j-1] \end{array} \right\}$$

↓
 $x_i = y_j$ $x_i \neq y_j$
 0 1

Sotto-problemi

- $X = x_1, x_2, \dots, x_n$ e $Y = y_1, y_2, \dots, y_m$
- Vogliamo usare Programmazione Dinamica
- **Cosa sono i sotto-problemi?**
- **Prefissi delle due sequenze**



Tabella

- $X = x_1, x_2, \dots, x_n$ e $Y = y_1, y_2, \dots, y_m$
- Vogliamo usare Programmazione Dinamica
- **Tabella di programmazione dinamica?**
 - **Matrice $M[i,j]$ che memorizza la edit distance dei prefissi di x e y lunghi i e j rispettivamente**
 - Quindi le righe della tabella rappresentano i caratteri di X e le colonne quelli di Y

Programmazione Dinamica

- Indichiamo con $p(i, j)$ la distanza locale dei caratteri x_i e y_j
- Se $x_i = y_j$, $p(i, j) = 0$
- Altrimenti, $p(i, j) = 1$

\vdots
 $i-1$ i - . . . -  +1

$$\begin{aligned}
 ed(\varepsilon, \underline{\underline{n}}) &= n \\
 ed(\underline{\underline{n}}, \varepsilon) &= n \\
 M[0, \underline{\underline{j}}] &= \underline{\underline{j}} \\
 M[\underline{\underline{i}}, 0] &= i
 \end{aligned}$$

	0	1	2	3	4	5	6
0	L	A	B	B	R	O	
1	A	1	1				
2	L	2					
3	B	3					
4	E	4					
5	R	5					
6	O	6					

The table illustrates the dynamic programming matrix for string matching. The columns represent the string x and the rows represent the string y . The first row and column are labeled with characters L, A, B, R, O. The matrix entries are labeled with the cost $p(i, j)$: 0 for matching characters, 1 for mismatching characters, and +1 for transitions between states. Red arrows indicate the path from state (0,0) to (6,6). The final result is highlighted in yellow.

Programmazione Dinamica – formulazione ricorsiva

- Come possiamo formulare ricorsivamente il calcolo di $M[i,j]$?

$x_1, x_2, \dots, x_i \text{ vs } y_1, y_2, \dots, y_j \rightarrow M[i-1, j-1] + p(i,j)$

$x_1, x_2, \dots, - \text{ vs } y_1, y_2, \dots, y_j \rightarrow M[i-1, j] + 1$

$x_1, x_2, \dots, x_i \text{ vs } y_1, y_2, \dots, - \rightarrow M[i, j-1] + 1$



Programmazione Dinamica – formulazione ricorsiva

- Come possiamo formulare ricorsivamente il calcolo di $M[i,j]$?

$$M[i,j] = \min \{ M[i, j-1] + 1 \text{ } \text{ (yellow)} \text{ }$$

$$M[i-1, j] + 1 \text{ } \text{ (blue)} \text{ }$$

$$M[i-1, j-1] + p(i,j) \text{ } \text{ (pink)} \text{ }$$

	0	1	2	3	4	5	6
0	x _i	0	1	2	3	4	5
1	y _j	L	A	B	B	R	O
2							
3							
4							
5							
6							

The table illustrates a dynamic programming grid for two sequences. The columns represent sequence x_i (values 0-5) and the rows represent sequence y_j (values 0-6). The first four rows show the initial state of the sequences. Rows 2, 3, and 4 are highlighted with colored cells: row 2 has a red cell at (2,2), row 3 has a yellow cell at (3,2) and a dark blue cell at (3,3), and row 4 has a light blue cell at (4,3).

Programmazione Dinamica – formulazione ricorsiva

- Come possiamo formulare ricorsivamente il calcolo di $M[i,j]$?

$$M[i,j] = \min \{ M[i, j-1] + 1 \\ M[i-1, j] + 1 \\ M[i-1, j-1] + p(i,j) \}$$

$\circ (n \cdot m)$

	x_i	0	1	2	3	4	5	6
	y_j	L	A	B	B	R	O	
0	0	1	2	3	4	5	6	
1	A	1	1	2	3	4	5	
2	L	2	1	2	2	3	4	5
3	B	3	2	2	2	2	3	4
4	E	4	3	3	3	3	3	4
5	R	5	4	4	4	4	3	4
6	O	6	5	5	5	5	4	3

Programmazione Dinamica — formulazione ricorsiva

- Come possiamo formulare ricorsivamente il calcolo di $M[i,j]$?

```
for (i=0, i≤ n, i++) {M[i,0]=i;} //inizializza la colonna 0  
for (j=0, j≤ m, j++) {M[0,j]=j;} //inizializza la riga 0  
for (i=1, i≤ n, i++)  
{ for (j=1, j≤ m, j++)  
    { if (X[i]==Y[j]) {p=0; } else {p=1; }  
        M[i,j]=min(M[i-1,j]+1,M[i-1,j-1]+p,M[i,j-1]+1); } }  
returnM[n,m];
```



Complessità: **O(nm)**

[REDACTED] come ricostruire
l'allineamento ottimo ?

	x_i	0	1	2	3	4	5	6
y_j	L	A	B	B	R	O		
0	0	1	2	3	4	5	6	
1	A	1	1	2	3	4	5	
2	L	2	1	2	3	4	5	
3	B	3	2	2	2	3	4	
4	E	4	3	3	3	3	3	4
5	R	5	4	4	4	4	3	4
6	O	6	5	5	5	5	4	3

Partizione di un insieme di interi (esercizio)

Esercizio Programmazione Dinamica

Partizione di un insieme di interi

- Supponiamo di avere **due** supporti di capacità s byte ciascuno, e n file **indivisibili** grandi in totale $2s$ byte
- **Problema:** è possibile dividere gli n file in **due gruppi** di s byte ciascuno?
 - Problema della **partizione**

Formulazione generale della partizione (**partition**):

- Insieme di interi positivi $A = \{a_0, a_1, \dots, a_{n-1}\}$
t.c. $\sum_{i=0,1,\dots,n-1} a_i = 2s$
- **Esiste** sottoinsieme $B = \{b_0, b_1, \dots, b_{k-1}\} \subseteq A$
t.c. $\sum_{j=0,1,\dots,k-1} b_j = s$?
 - Soluzione **banale**: tempo **esponenziale** $O(2^n n)$ se generiamo tutti i sottoinsiemi

Programmazione dinamica

Seguiamo le solite regole:

1. definiamo per $0 \leq i \leq n$ e $0 \leq j \leq s$

$T(i,j) = \text{true sse}$
 $\text{esiste } S \subseteq \{a_0, a_1, \dots, a_{i-1}\} \text{ t.c. } \sum_{y \in S} y = j$

2. **Domanda:** come possiamo formulare il problema originario??

1. $T(n,s)$

[convenzione: $\{a_0, a_1, \dots, a_{-1}\} = \text{insieme vuoto}$]

Programmazione Dinamica

- **Domanda:** quali sono i sotto-problemi elementari (quelli per cui possiamo dare subito la soluzione)?
 - $T(0,0)=\text{true}$
 - $T(0,j)=\text{false}, 0 < j \leq s$
- Cerchiamo ora di definire il **caso generale** $T(i,j)$ in termini dei casi aventi valori i e j più piccoli (programmazione dinamica....)
 - Seguiamo l'approccio seguito per il **problema LCS**
 - a_{i-1} **fa parte** della soluzione immediatamente più piccola **o no?**

Programmazione Dinamica

- Cerchiamo ora di definire il **caso generale** $T(i,j)$ in termini dei casi aventi valori i e j più piccoli (programmazione dinamica....)
 - Seguiamo l'approccio seguito per il **problema LCS**
 - a_{i-1} **fa parte** della soluzione o no?
 - **Provate ad analizzare i casi**

esiste già un sotto-insieme
(che non contiene a_{i-1}) la cui
somma è j

non esiste già un sotto-
insieme la cui somma è j , e
che permette l'aggiunta di a_{i-1}

else

Formulazione ricorsiva

$$T(i, j) = \begin{cases} \text{true} & \text{se } i = 0 \text{ e } j = 0 \\ \text{true} & \text{se } i > 0 \text{ e } T(i - 1, j) = \text{true} \\ \text{true} & \text{se } i > 0, j \geq a_{i-1} \text{ e } T(i - 1, j - a_{i-1}) = \text{true} \\ \text{false} & \text{altrimenti} \end{cases}$$

se l'insieme è vuoto allora la somma $j=0$ è l'unica possibile

Formulazione ricorsiva

$$T(i, j) = \begin{cases} \text{true} & \text{se } i = 0 \text{ e } j = 0 \\ \text{true} & \text{se } i > 0 \text{ e } T(i - 1, j) = \text{true} \\ \text{true} & \text{se } i > 0, j \geq a_{i-1} \text{ e } T(i - 1, j - a_{i-1}) = \text{true} \\ \text{false} & \text{altrimenti} \end{cases}$$

Caso 1. Il sottoinsieme di $\{a_0, a_1, \dots, a_{i-1}\}$ la cui somma è pari a j **non** include a_{i-1} : tale insieme è dunque sottoinsieme anche di $\{a_0, a_1, \dots, a_{i-2}\}$ e vale $T(i-1,j)=\text{true}$

Formulazione ricorsiva

$$T(i, j) = \begin{cases} \text{true} & \text{se } i = 0 \text{ e } j = 0 \\ \text{true} & \text{se } i > 0 \text{ e } T(i - 1, j) = \text{true} \\ \text{true} & \text{se } i > 0, j \geq a_{i-1} \text{ e } T(i - 1, j - a_{i-1}) = \text{true} \\ \text{false} & \text{altrimenti} \end{cases}$$

Caso 2. Il sottoinsieme di $\{a_0, a_1, \dots, a_{i-1}\}$ la cui somma è pari a j **include** a_{i-1} : esiste quindi in $\{a_0, a_1, \dots, a_{i-2}\}$ un sottoinsieme di somma pari a $j - a_{i-1}$ (ovviamente se $j \geq a_{i-1}$) e vale $T(i-1, j-a_{i-1})=\text{true}$

Programmazione dinamica per partizione

Tabella di programmazione dinamica di taglia $(n+1) \times (s+1)$

$\text{parti}[i][j] = T(i,j)$

Ordine riempimento: per righe

Pseudocodice

```
1 Partizione( a ):           <pre: a è un array di n interi positivi>
2     FOR (i = 0; i <= n; i = i+1)
3         FOR (j = 0; j <= s; j = j+1) {
4             parti[i][j] = FALSE;
5         }
6         parti[0][0] = TRUE;
7         FOR (i = 1; i <= n; i = i+1)
8             FOR (j = 0; j <= s; j = j+1) {
9                 IF (parti[i-1][j]) {
10                     parti[i][j] = TRUE;
11                 }
12                 IF (j >= a[i-1] && parti[i-1][j-a[i-1]]) {
13                     parti[i][j] = TRUE;
14                 }
15             }
16     RETURN parti[n][s];
```

$$T(i, j) = \begin{cases} \text{true} & \text{se } i = 0 \text{ e } j = 0 \\ \text{true} & \text{se } i > 0 \text{ e } T(i-1, j) = \text{true} \\ \text{true} & \text{se } i > 0, j \geq a_{i-1} \text{ e } T(i-1, j - a_{i-1}) = \text{true} \\ \text{false} & \text{altrimenti} \end{cases}$$

Analisi di complessità per partizione

- Tempo $O(ns)$
- Spazio $O(ns)$: $O(s)$ usando solo le ultime due righe compilate iterativamente

Il problema dello zaino 0-1 ‘intero’

Avete pensato al
rod-cutting problem ?
.... well

Il problema dello zaino ‘intero’

Abbiamo uno zaino e vogliamo riempirlo con
un pò di oggetti

La nostra forza è (ahimè) limitata, e quindi
riusciamo a trasportare al **massimo un**
certo peso

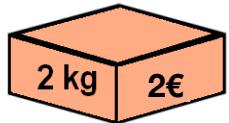
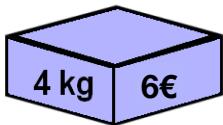
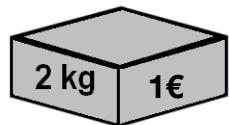
Inoltre, ogni oggetto ha un certo **valore**

Quindi vogliamo trasportare oggetti in modo
da **massimizzare il valore** del trasporto



Il problema dello zaino ‘intero’

(simile al rod-cut ma.....
"divisioni" date!)



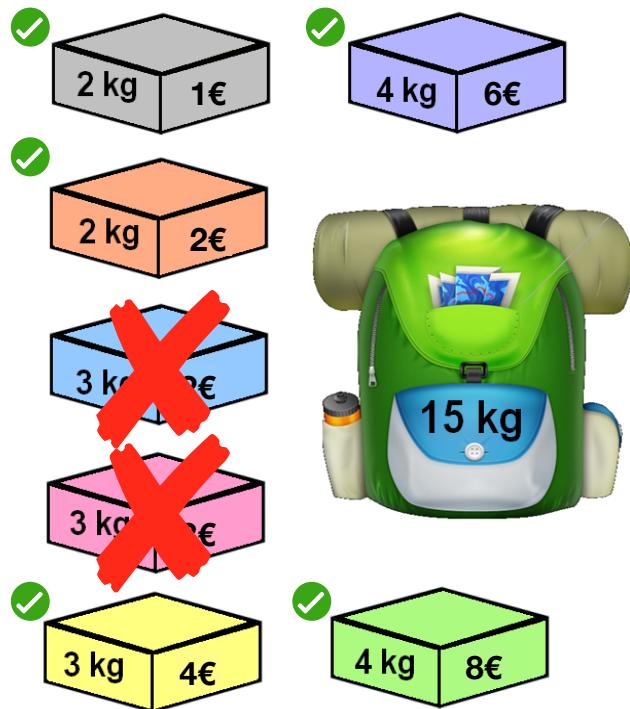
- Ogni oggetto ha peso e valore assegnati

- Massimizzare il valore

- Massimo peso trasportabile: 15 kg

- Padroneggiando ormai la Programmazione Dinamica, la usiamo per risolvere il problema!!!!

Soluzione????



Il problema dello zaino ‘intero’

- Esiste la versione ‘non intera’ del problema, in cui è possibile prendere solo una parte dell’oggetto
- Nella formulazione 0-1 è possibile prendere o non prendere
- Da qui il nome ‘**intero**’ o anche **0-1**

Il problema dello zaino ‘intero’

- **Obiettivo:** **massimizzare il valore dello zaino** che può contenere al più peso W , scegliendo oggetti da una lista $I_0, I_1, \dots I_{n-1}$.
- Ogni oggetto ha 2 attributi:
 - **Valore** – sia v_i per l'oggetto I_i
 - **Peso** – sia w_i per l'oggetto I_i

Al solito....

- Forza brut(t)a
 - Esaminare tutti i possibili **2^n** sottoinsiemi di n oggetti e scegliere il sottoinsieme con un peso ammissibile che massimizza il valore
 - Non ci piace, quindi proviamo con la Programmazione Dinamica

Il problema dello zaino ‘intero’

- Proviamo a risolvere il problema in termini di sotto-problemi.
Caratterizziamoli:
 - Sia S_k la soluzione ottima per $\{I_0, I_1, \dots, I_k\}$
 - Quello che può accadere è che l’ottimo per $\{I_0, I_1, \dots, I_{k+1}\}$ potrebbe non derivare dalla soluzione ottima per $\{I_0, I_1, \dots, I_k\}$
 - In altre parole, la soluzione S_{k+1} potrebbe NON contenere elementi di S_k

$k-1$ elementi \rightarrow ottima per pesi
0... w

$I_0, I_1, \dots, I_{k-1}, I_k$

• I_k si $\Rightarrow I_k \oplus I_0, \dots, I_{k-1}$ $\stackrel{w_k}{\leftarrow}$

• I_k no \Rightarrow sol. ottima in I_0, \dots, I_{k-1}

Il problema dello zaino 'intero'

Peso Massimo: 20

- L'ottimo per $\{I_0, I_1, I_2\}$ è $S_2 = \{I_0, I_1, I_2\}$
- MA l'ottimo per $\{I_0, I_1, I_2, I_3\}$ è $S_3 = \{I_0, I_2, I_3\}$.
- Si noti che S_3 NON nasce da S_2
 - Piuttosto è costruita su $\{I_0, I_2\}$ che è l'ottimo per $\{I_0, I_1, I_2\}$ quando il peso massimo è 12

Elemento	Peso	Valore
I_0	3	10
I_1	8	4
I_2	9	9
I_3	8	11

$$B[0, w] = 0$$

$$B[k, 0] = 0$$

Il problema dello zaino ‘intero’ – formulazione ricorsiva

- L’ottimo S_k per i primi k elementi, per un peso massimo w , o contiene l’elemento k oppure no

$$B[k, w] = \begin{cases} B[k - 1, w] & \text{se } w_k > w \\ \max\{B[k - 1, w], B[k - 1, w - w_k] + v_k\} & \text{altrimenti} \end{cases}$$

$0 \leq w \leq W$

Annotations:

- $B[k, w]$: *# deve per zaino*
- $B[k - 1, w]$: *non e' utile*
- $B[k - 1, w - w_k] + v_k$: *e' utile*
- $w_k \leq w$

- Primo caso:** $w_k > w$

- L’elemento k non può essere parte della soluzione, altrimenti il peso totale supererebbe w

- Secondo caso:** $w_k \leq w$

- L’elemento k potrebbe essere parte della soluzione finale

Avendo inserito l’oggetto k -esimo, il valore è incrementato di v_k , ma il peso a disposizione per aggiungere oggetti è rimasto il peso totale disponibile w meno il peso dell’oggetto w_k

$$B[k, w] = \begin{cases} B[k - 1, w] & \text{se } w_k > w \\ \max\{B[k - 1, w], B[k - 1, w - w_k] + v_k\} & \text{altrimenti} \end{cases}$$

Il problema dello zaino ‘intero’ – soluzione

Cosa usiamo come **Tabella di Programmazione Dinamica?**

Matrice B: n x W

```
// Inizializzazione
for w = 0 to W
    B[0,w] = 0
for i = 1 to n
    B[i,0] = 0
```

```
if w_i <= w //elemento i potrebbe essere nella soluzione
    if v_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = v_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w
```

Il problema dello zaino ‘intero’ – esempio

$\omega \leftarrow$

(2,3), (3,4), (4,5), (5,6)

i / w	0	1	2	3	4	5
0						

$W=5$

// Inizializzazione

for $w = 0$ to W

$B[0,w] = 0$

for $i = 1$ to n

$B[i,0] = 0$

Il problema dello zaino ‘intero’ – esempio

w s

(2,3), (3,4), (4,5), (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

// Inizializzazione

for w = 0 to W

B[0,w] = 0

for i = 1 to n

B[i,0] = 0

(w, v)

Il problema dello zaino ‘intero’ – esempio

(2,3), (3,4), (4,5), (5,6)

riempio
per righe
 w/v
2,3
3,4
4,5
5,6

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0			
2	0					
3	0					
4	0					

$$\begin{aligned} i &= 1 \\ v_i &= 3 \\ w_i &= 2 \\ w &= 1 \\ w - w_i &= -1 \end{aligned}$$

if $w_i \leq w$ //elemento i potrebbe essere nella soluzione

if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = v_i + B[i-1, w-w_i]$

else

$B[i, w] = B[i-1, w]$

else $B[i, w] = B[i-1, w]$ // $w_i > w$

$3 + B[i-1, w-2]$



Il problema dello zaino ‘intero’ – esempio

w, v

(2,3), (3,4), (4,5), (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

ω/σ
2,3
3,4
4,5
5,6

$i = 1$
 $v_i = 3$
 $w_i = 2$ jump indietro
 $w = 2$
 $w - w_i = 0$

if $w_i \leq w$ //elemento i potrebbe essere nella soluzione

if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else $B[i, w] = B[i-1, w]$ // $w_i > w$

Il problema dello zaino ‘intero’ – esempio

ω, ς

(2,3), (3,4), (4,5), (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0					
3	0					
4	0					

$i = 1$
 $v_i = 3$
 $w_i = 2$ jump indietro
 $w = 3$
 $w - w_i = 1$

if $w_i \leq w$ //elemento i potrebbe essere nella soluzione

if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else $B[i, w] = B[i-1, w]$ // $w_i > w$

Il problema dello zaino ‘intero’ – esempio

ω, σ

(2,3), (3,4), (4,5), (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0					
3	0					
4	0					
5	0					

$i = 1$
 $v_i = 3$
 $w_i = 2$
 $w = 4$
 $w - w_i = 2$

if $w_i \leq w$ //elemento i potrebbe essere nella soluzione

if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else $B[i, w] = B[i-1, w]$ // $w_i > w$

Il problema dello zaino ‘intero’ – esempio

w, k

(2,3), (3,4), (4,5), (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

ω/σ
 2,3
 3,4
 4,5
 5,6

$i = 1$
 $v_i = 3$
 $w_i = 2$
 $w = 5$
 $w - w_i = 3$

if $w_i \leq w$ //elemento i potrebbe essere nella soluzione

if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else $B[i, w] = B[i-1, w]$ // $w_i > w$

Il problema dello zaino ‘intero’ – esempio

ω, ς
 $(2,3), (3,4), (4,5), (5,6)$

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					

$\omega/5$
 $2,3$
 $3,4$
 $4,5$
 $5,6$

$i = 2$
 $v_i = 4$
 $w_i = 3$
 $w = 1$
 $w - w_i = -2$

if $w_i \leq w$ //elemento i potrebbe essere nella soluzione

if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = v_i + B[i-1, w-w_i]$

else

$B[i, w] = B[i-1, w]$

else **$B[i, w] = B[i-1, w]$** // $w_i > w$

ω, ζ

Il problema dello zaino ‘intero’ – esempio

(2,3), **(3,4)**, (4,5), (5,6)

ω/ζ $3,4$

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	0		
3	0					
4	0					

$i = 2$
 $v_i = 4$
 $w_i = 3$
 $w = 2$
 $w - w_i = -1$

4

if $w_i \leq w$ //elemento i potrebbe essere nella soluzione

if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w-w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else **$B[i, w] = B[i-1, w]$** // $w_i > w$

ω, ζ

Il problema dello zaino ‘intero’ – esempio

(2,3), **(3,4)**, (4,5), (5,6)

$\omega/5$ $3,4$

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		
3	0					
4	0					

$i = 2$
 $v_i = 4$
 $w_i = 3$ jump indicato
 $w = 3$
 $w - w_i = 0$

if $w_i \leq w$ //elemento i potrebbe essere nella soluzione

 if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = v_i + B[i-1, w-w_i]$

 else

$B[i, w] = B[i-1, w]$

else $B[i, w] = B[i-1, w]$ // $w_i > w$

ω, σ

Il problema dello zaino ‘intero’ – esempio

(2,3), **(3,4)**, (4,5), (5,6)

ω/σ

$\underline{\underline{=3,4=}}$

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0					
4	0					

$i = 2$
 $v_i = 4$
 $w_i = 3$
 $w = 4$
 $w - w_i = 1$

$B[i-1, 5-3]$

if $w_i \leq w$ //elemento i potrebbe essere nella soluzione

if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w-w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else $B[i, w] = B[i-1, w]$ // $w_i > w$

ω, δ

Il problema dello zaino ‘intero’ – esempio

(2,3), **(3,4)**, (4,5), (5,6)

3,4

i / w	0	1	2	3	4	5	
0	0	0	0	0	0	0	
1	0	0	3	3	3	3	
2	0	0	3	4	4	7	
3	0						
4	0						

$i = 2$
 $v_i = 4$
 $w_i = 3$
 $w = 5$
 $w - w_i = 2$

if $w_i \leq w$ //elemento i potrebbe essere nella soluzione

if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else $B[i, w] = B[i-1, w]$ // $w_i > w$

Il problema dello zaino ‘intero’ – esempio

(2,3), (3,4), **(4,5)**, (5,6)

ω/σ

4,5

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4		
4	0					

i = 3
 $v_i = 5$
 $w_i = 4$
 $w = 1..3$
 $w - w_i = -3..-1$

if $w_i \leq w$ //elemento i potrebbe essere nella soluzione

if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = v_i + B[i-1, w-w_i]$

else

$B[i, w] = B[i-1, w]$

else **B[i, w] = B[i-1, w]** // $w_i > w$

ω, σ

Il problema dello zaino ‘intero’ – esempio

(2,3), (3,4), **(4,5)**, (5,6)

$\omega/5$
 $4,5$

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

$i = 3$
 $v_i = 5$
 $w_i = 4$
 $w = 4$
 $w - w_i = 0$

if $w_i \leq w$ //elemento i potrebbe essere nella soluzione

if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else $B[i, w] = B[i-1, w]$ // $w_i > w$

ω, σ

Il problema dello zaino ‘intero’ – esempio

(2,3), (3,4), **(4,5)**, (5,6)

ω/σ
4,5

i / w	0	1	2	3	4	5	
0	0	0	0	0	0	0	
1	0	0	3	3	3	3	
2	0	0	3	4	4	7	
3	0	0	3	4	5	7	
4	0						

$i = 3$
 $v_i = 5$
 $w_i = 4$
 $w = 5$
 $w - w_i = 1$

if $w_i \leq w$ //elemento i potrebbe essere nella soluzione

 if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = v_i + B[i-1, w-w_i]$

 else

$B[i, w] = B[i-1, w]$

else $B[i, w] = B[i-1, w]$ // $w_i > w$

w, σ

Il problema dello zaino ‘intero’ – esempio

(2,3), (3,4), (4,5), **(5,6)**

ω, ς
5,6

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	

$$\begin{aligned}
 i &= 4 \\
 v_i &= 6 \\
 w_i &= 5 \\
 w &= 1..4 \\
 w - w_i &= -4..-1
 \end{aligned}$$

if $w_i \leq w$ //elemento i potrebbe essere nella soluzione

if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w-w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else **B[i,w] = B[i-1,w]** // $w_i > w$

ω, ζ

Il problema dello zaino ‘intero’ – esempio

(2,3), (3,4), (4,5), **(5,6)**

ω, ζ

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 4$
 $v_i = 6$
 $w_i = 5$
 $w = 5$
 $w - w_i = 0$

if $w_i \leq w$ //elemento i potrebbe essere nella soluzione

if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = v_i + B[i-1, w-w_i]$

else

$B[i, w] = B[i-1, w]$

else $B[i, w] = B[i-1, w]$ // $w_i > w$

Il problema dello zaino ‘intero’ – esempio

(2,3), (3,4), (4,5), **(5,6)**

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

FINITO!!!!

Il valore massimo per questo zaino è 7!!!!

Il problema dello zaino ‘intero’

- Questo algoritmo trova solo il massimo valore possibile dato un peso massimo
 - Valore massimo in $B[n, W]$
- Per conoscere con quali **elementi** posso raggiungere il valore massimo, dobbiamo in qualche modo tracciarli nelle tabella

$$w_i \leq W \longrightarrow O(\log_2 w_i) \text{ bit}$$
$$O(\log_2 W) \text{ bit}$$

$$\text{dim.} = O(n \log W)$$

input

$$O(n \cdot W)$$
$$n \cdot 2^{\log_2 W}$$

pseudo-poly

Il problema dello zaino ‘intero’ – trovare gli elementi dell’ottimo

- Siano $i = n$ e $k = W$

if $B[i, k] \neq B[i-1, k]$ then

marca l’elemento i -esimo come parte dello zaino

$i = i-1, k = k-w_i$

else

$i = i-1 //$ L’elemento i -esimo non è nello zaino

$$B[k, w] = \begin{cases} B[k - 1, w] & \text{se } w_k > w \\ \max\{B[k - 1, w], B[k - 1, w - w_k] + v_k\} & \text{altrimenti} \end{cases}$$

Elementi: Zaino:

- | |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

Trovare gli elementi dell'ottimo

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$$i = 4$$

$$k = 5$$

$$v_i = 6$$

$$w_i = 5$$

$$B[i,k] = 7$$

$$B[i-1,k] = 7$$

$$i = n, k = W$$

while $i, k > 0$

if $B[i, k] \neq B[i-1, k]$ then

marca i^{th} -esimo elemento come nello zaino

$$i = i-1, k = k-w_i$$

else

$$i = i-1$$

Elementi: Zaino:

- | |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

Trovare gli elementi dell'ottimo

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$$i = 3$$

$$k = 5$$

$$v_i = 5$$

$$w_i = 4$$

$$B[i,k] = 7$$

$$B[i-1,k] = 7$$

$$i = n, k = W$$

while $i, k > 0$

if $B[i, k] \neq B[i-1, k]$ then

marca i^{th} -esimo elemento come nello zaino

$i = i-1, k = k-w_i$

else

$i = i-1$

Elementi:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Zaino:

Elem 2

Trovare gli elementi dell'ottimo

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

ω, σ
3, 4

i = 2

k = 5

v_i = 4

w_i = 3

B[i,k] = 7

B[i-1,k] = 3

k - w_i = 2

i = n , k = W

while i, k > 0

if B[i, k] ≠ B[i-1, k] then

marca ith -esimo elemento come nello zaino

i = i-1, k = k-w_i

else

i = i-1

Trovare gli elementi dell'ottimo

Elementi:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

Zaino:

- Elem 2**
- Elem 1**

ω, ς
2, 3

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$$i = 1$$

$$k = 2$$

$$v_i = 3$$

$$w_i = 2$$

$$B[i,k] = 3$$

$$B[i-1,k] = 0$$

$$k - w_i = 0$$

$$i = n, k = W$$

while $i, k > 0$

if $B[i, k] \neq B[i-1, k]$ then

marca i^{th} -esimo elemento come nello zaino

$$i = i-1, k = k-w_i$$

else

$$i = i-1$$

Trovare gli elementi dell'ottimo

Elementi:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

Knapsack:

Elem 2
Elem 1

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$$i = 1$$

$$k = 2$$

$$v_i = 3$$

$$w_i = 2$$

$$B[i,k] = 3$$

$$B[i-1,k] = 0$$

$$k - w_i = 0$$

$i = 0, k = 0$, e abbiamo **finito!**

Lo zaino ottimo contiene:

Elemento 1 e Elemento 2

Il problema dello zaino ‘intero’ – Tempo

for w = 0 to W
 B[0,w] = 0

O(W)

for i = 1 to n
 B[i,0] = 0

O(n)

for i = 1 to n
 for w = 0 to W
 < >

Ripeti n volte
O(W)

Tempo di esecuzione?

O($n * W$)

Il problema dello zaino ‘intero’ – Esercizio

Riempire la tabella di programmazione dinamica per questi elementi

Determinare il valore ottimo e gli elementi che determinano l’ottimo

4 kg, 6€

1 kg, 3€



6 kg, 9€

3 kg, 5€

2 kg, 4€

RAM

$$O(n \cdot w) = O(n \cdot 2^{\lg_2 w})$$

clear
max specifici zeri

$$\text{dove input} = O(u+1) = \Theta(u)$$



$$O(\lg_2 w) \text{ bit}$$

Pseudo-polynomialità

$$\# \text{bit input} = O((u+1) \lg_2 w)$$

Pseudo-polinomialità

- La complessità dell'algoritmo dello Zaino 0-1 è $O(nW)$
- Quindi il problema dipende anche da W
 - Cosa succede se W è molto **grande??**
 - La dimensione della tabella di programmazione dinamica **aumenta, e quindi la complessità**
 - In altre parole, il problema è **polinomiale** in n e W , ma **non necessariamente nella dimensione** dei **dati ingresso** (W)

Dimensione dell'input:
ho n oggetti per cui
fornisco n valori e n
pesi, cioè $2n$ valori e
quindi la dimensione è
 $O(n)$

Valore dell'input, W è
proprio il peso. La sua
dimensione è il numero
di bit necessari a
rappresentarlo

Pseudo-polinomialità

- Domanda: quanti bit occorrono per rappresentare W ?
 - $k = O(\log W)$
- Domanda: quindi la vera complessità del problema dello zaino è....
 - $O(nW) = O(n2^k)$
 - Costo esponenziale rispetto alla dimensione dell'input
 - In altre parole, $O(nW)$ è una complessità polinomiale soltanto se $W = O(n^c)$ per una costante $c > 0$
- Per questo motivi questo tipo di algoritmi viene detto pseudo-polinomiale
 - *Polinomiale solo se si usano interi piccoli rispetto a n*

Algoritmi Greedy



Algoritmi Greedy

- Simile alla programmazione dinamica, ma con un approccio più **semplice**
 - Utilizzato (come la Programmazione Dinamica) per problemi di ottimizzazione
- **Idea:** Quando c'è da fare una scelta, **fa quella che è meglio ORA**
 - Effettua scelte ottime **locali** nella speranza che siano ottime per il problema globale
-e infatti non sempre conducono a scelte ottime globali

Problema dello zaino frazionario

- Peso massimo: W
- Ci sono n elementi, ognuno con un valore v_i e peso w_i
- **Obiettivo:**
 - trovare x_i tali che per ogni $0 \leq x_i \leq 1$, $i = 1, 2, \dots, n$

$$\sum w_i x_i \leq W \text{ e}$$

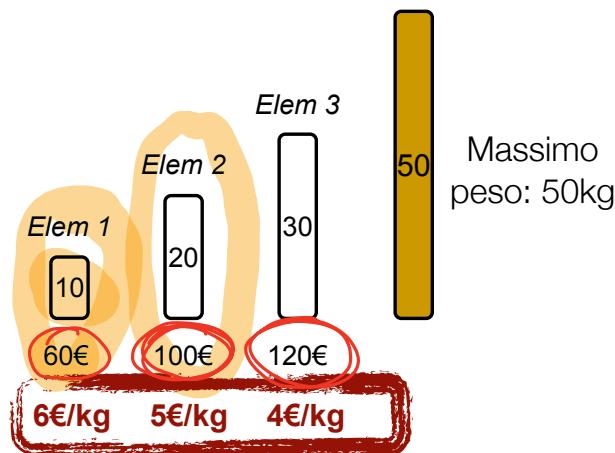
$\sum x_i v_i$ è massimo

Possiamo prendere
frazioni degli oggetti

$$d_i = \frac{v_i}{w_i} - O(u)$$

$$\text{sort} = O(u \cdot \log u)$$

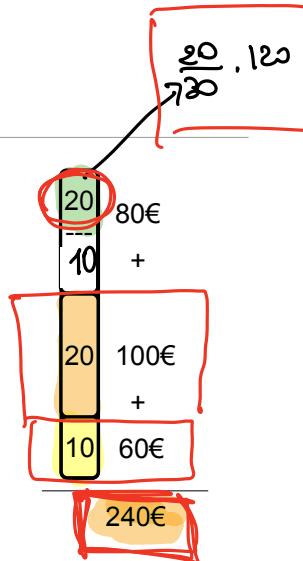
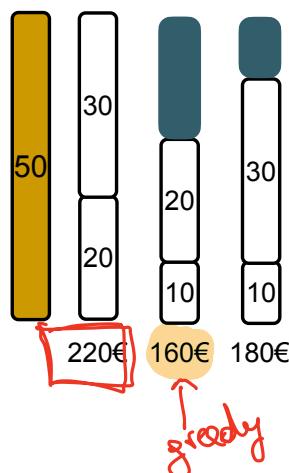
Problema dello zaino frazionario — esempio



Al supermercato quale scegliereste?

Elemento con il miglior rapporto costo al kg!

Versione Intera



Versione Frazionaria

Strategia Greedy: scegliere l'ottimo locale....quale è l'ottimo locale?

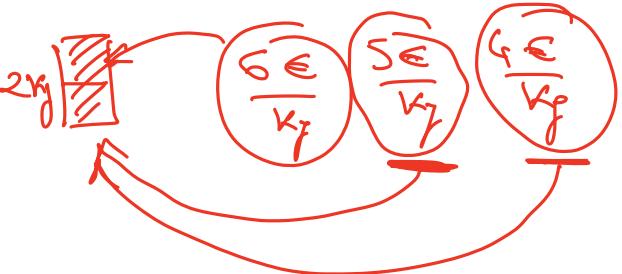
Problema dello zaino frazionario

Strategia Greedy:

- Scegli elemento con il massimo valore per peso v_i/w_i
- Se la quantità di quell'elemento termina, e lo zaino non è ancora al massimo peso possibile: **prendi quanto possibile dal prossimo elemento** con il migliore valore per peso
- Tipicamente si ordinano gli elementi in base al valore per peso:

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \dots \geq \frac{v_n}{w_n}$$





Problema dello zaino frazionario

Zaino Frazionario ($W, v[n], w[n]$) → peso "rimasto"

1. While $W > 0$ e ci sono elementi da aggiungere

2. scegli elemento con il massimo v_i/w_i

3. $x_i \leftarrow \min(1, W/w_i)$ intero $x_i = 1 \Leftrightarrow \frac{W}{w_i} \geq 1 \Leftrightarrow w_i \leq W$

4. rimuovi elemento i dalla lista

5. $W \leftarrow W - x_i w_i$ $w_i \text{ se } w_i \leq W$
 $W \text{ se } w_i > W$

- **Tempo:** $\Theta(n)$ se gli elementi sono già ordinati, altrimenti $\Theta(n \lg n)$

$$\frac{W}{w_i} : w_i$$

Scheduling di attività (esercizio)

Greedy

Scheduling di attività

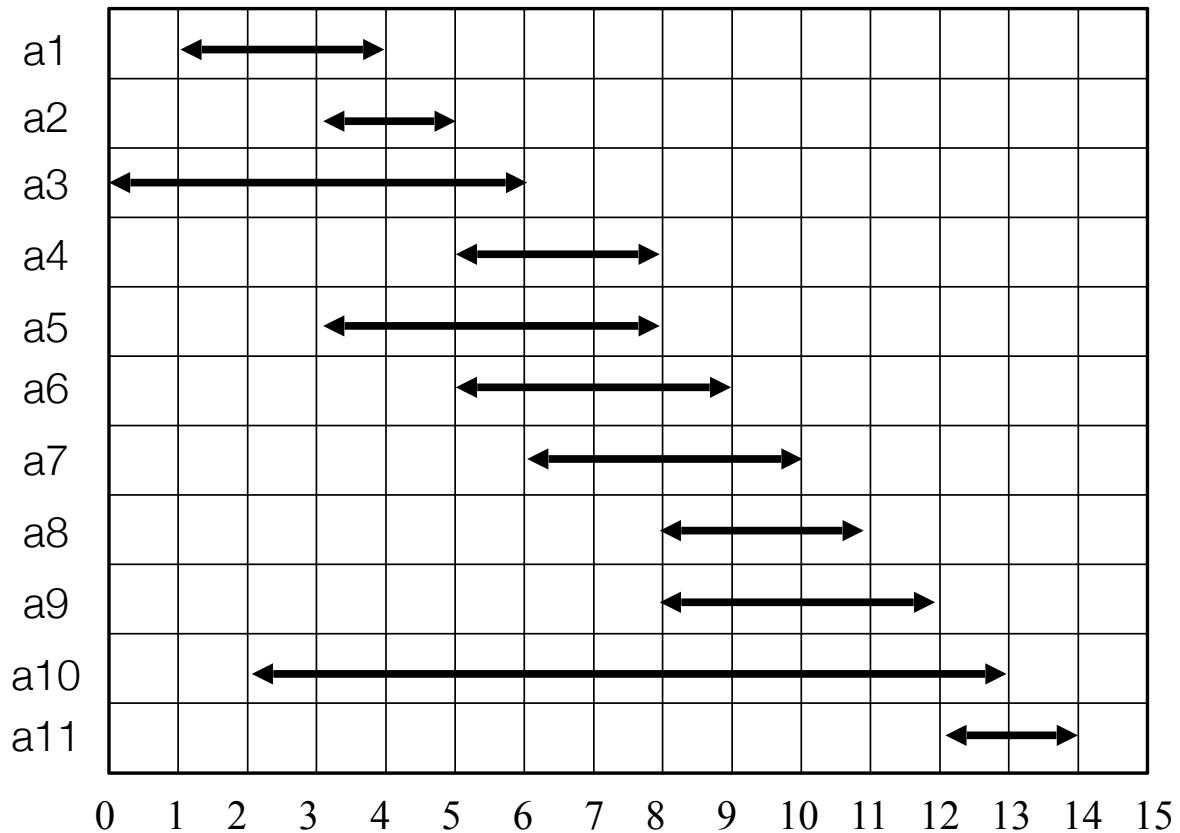
- **Input: insieme di attività $S = \{a_1, \dots, a_n\}$**
- Ogni attività ha un tempo di inizio e uno di fine
 - $a_i = (s_i, f_i)$
- Due attività sono compatibili se e solo se i loro intervalli di esecuzione non si sovrappongono
- **Output: insieme massimo di attività mutuamente compatibili**

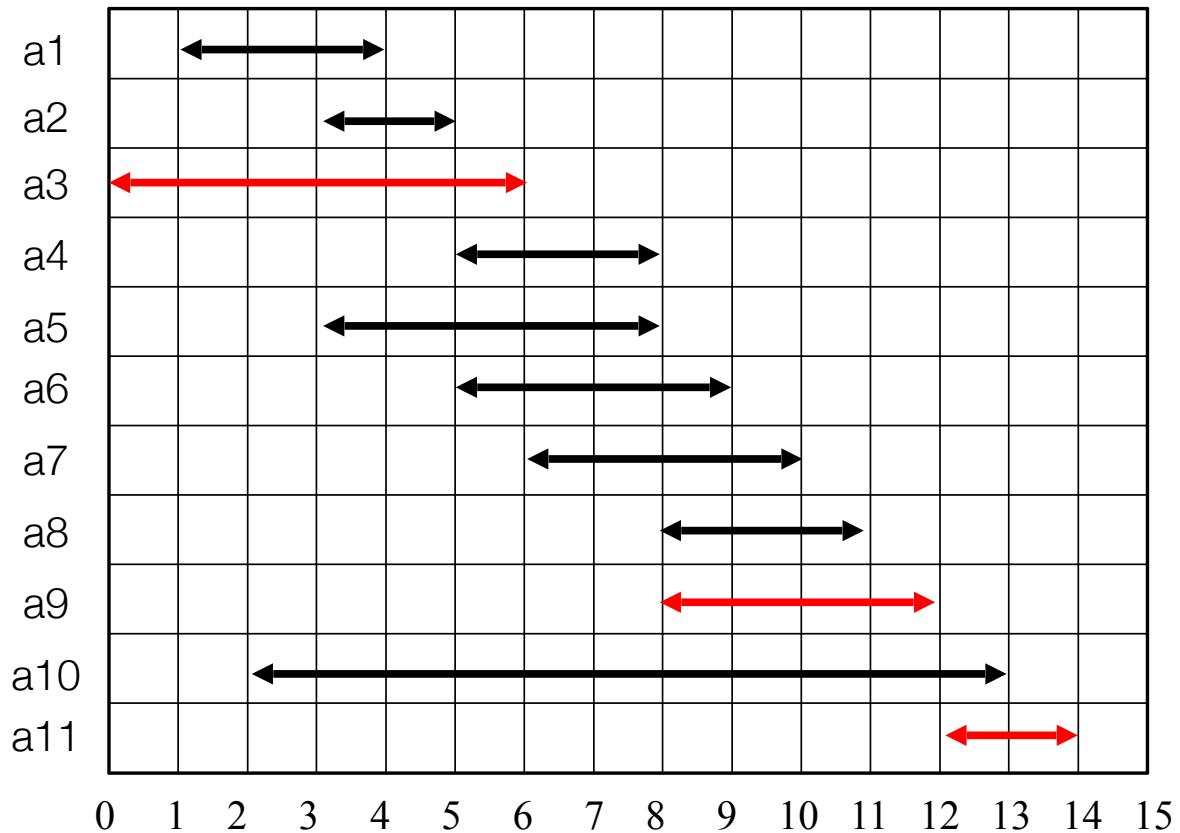
Scheduling di attività

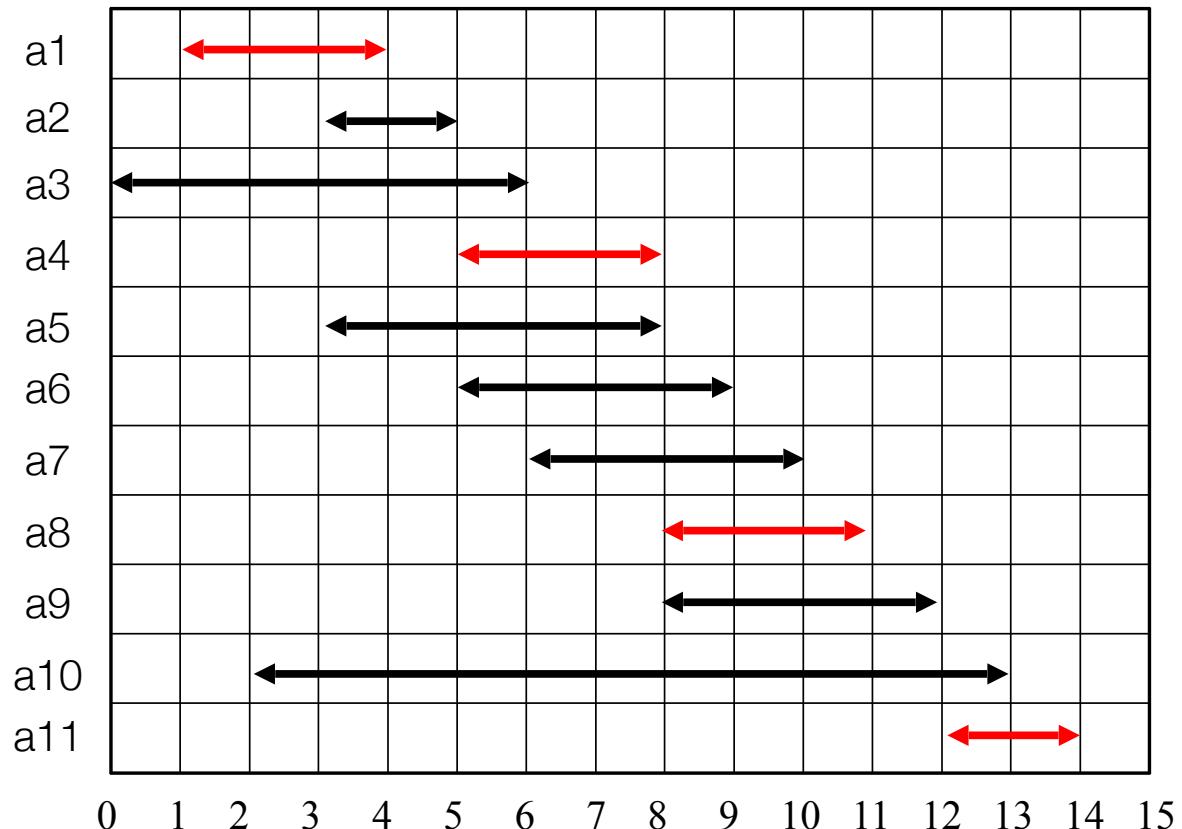
- **Esempio**

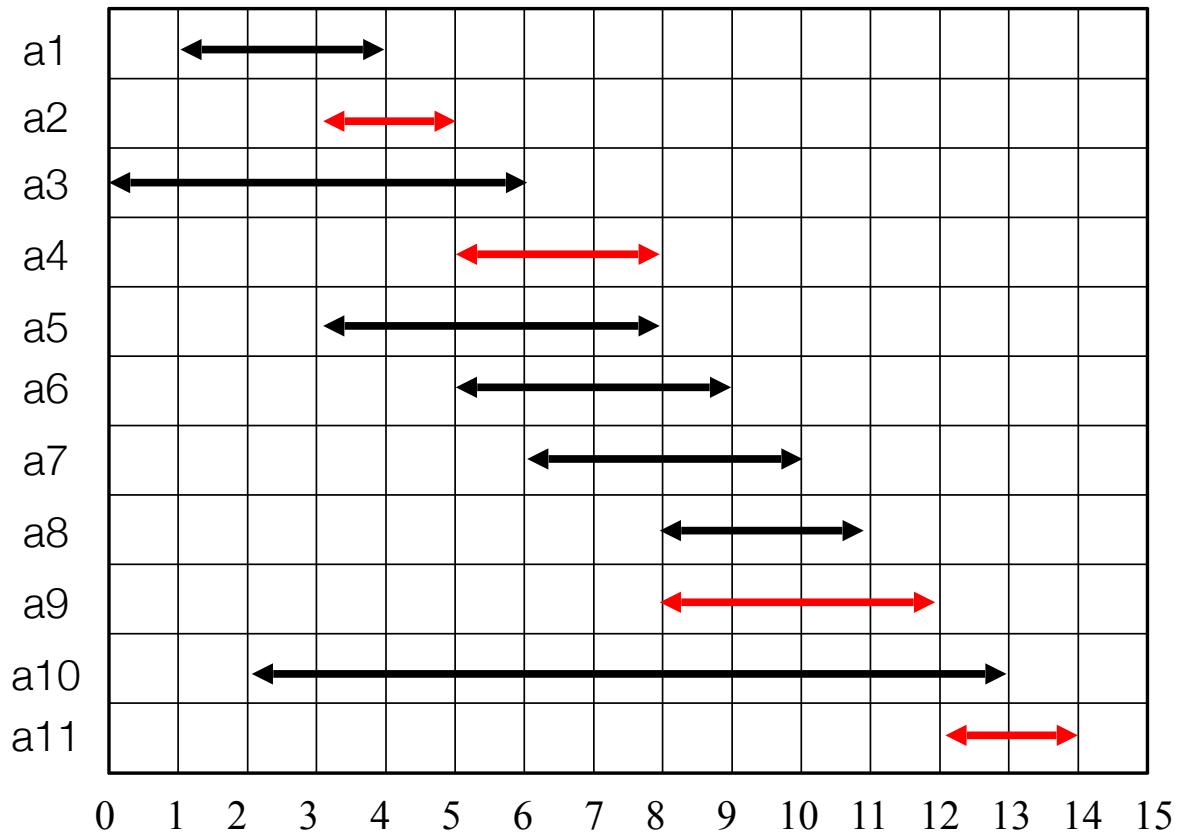
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- Qual è il massimo numero di attività che possono essere completate rispettando il vincolo di non sovrapposizione?
 - $\{a_3, a_9, a_{11}\}$
 - Ma anche $\{a_1, a_4, a_8, a_{11}\}$, che è un insieme più grande
 - Ma non unico.... $\{a_2, a_4, a_9, a_{11}\}$



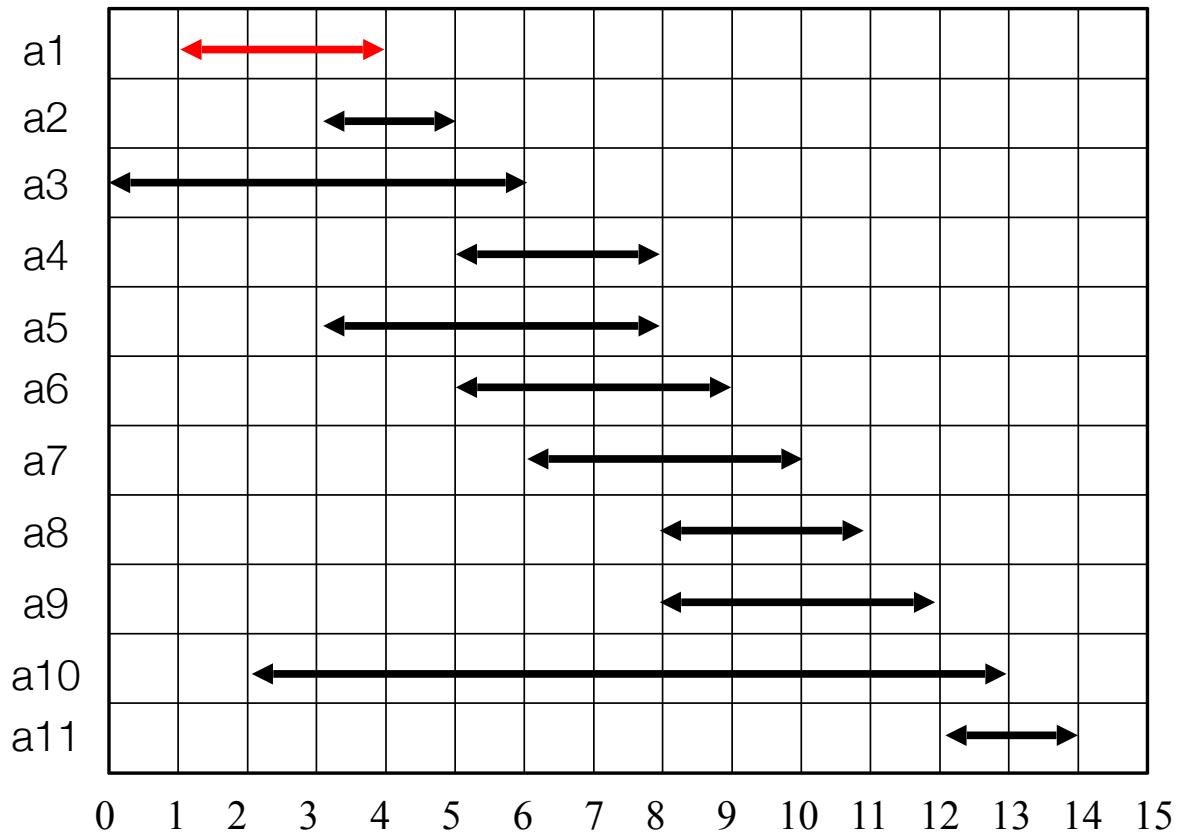


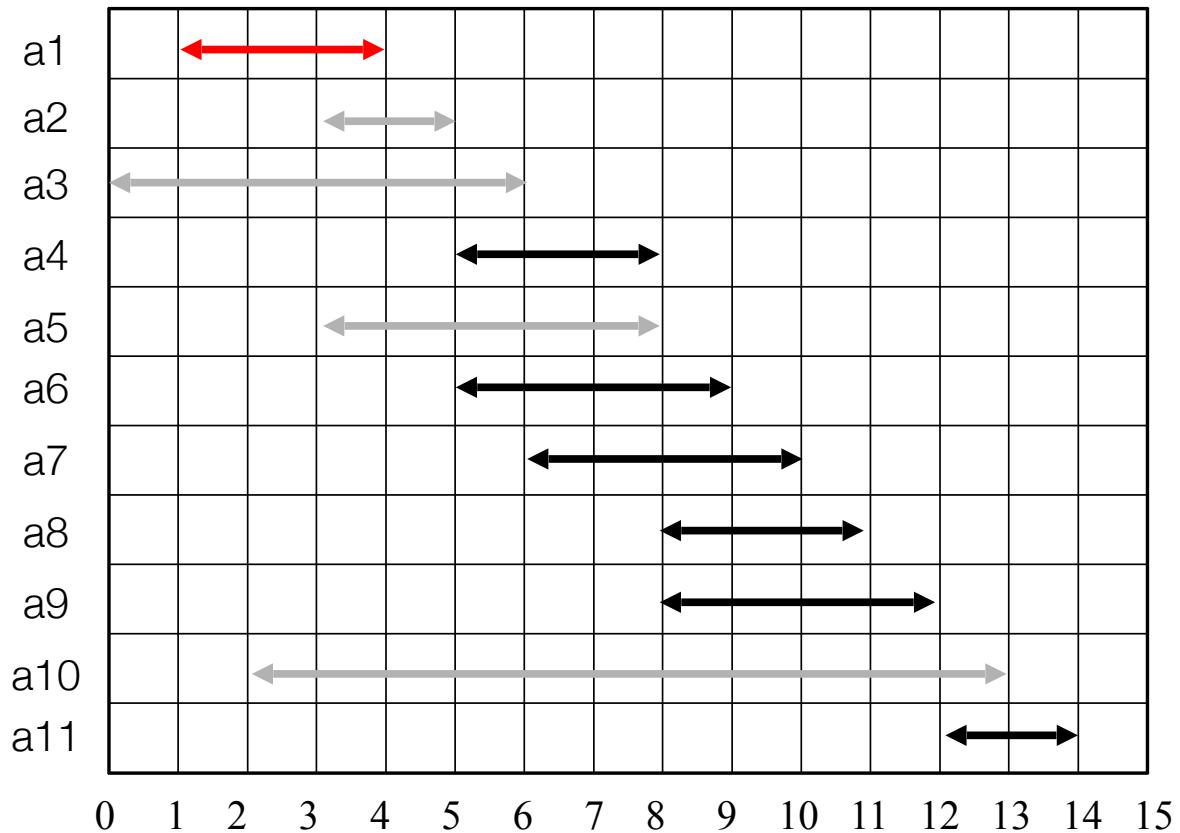


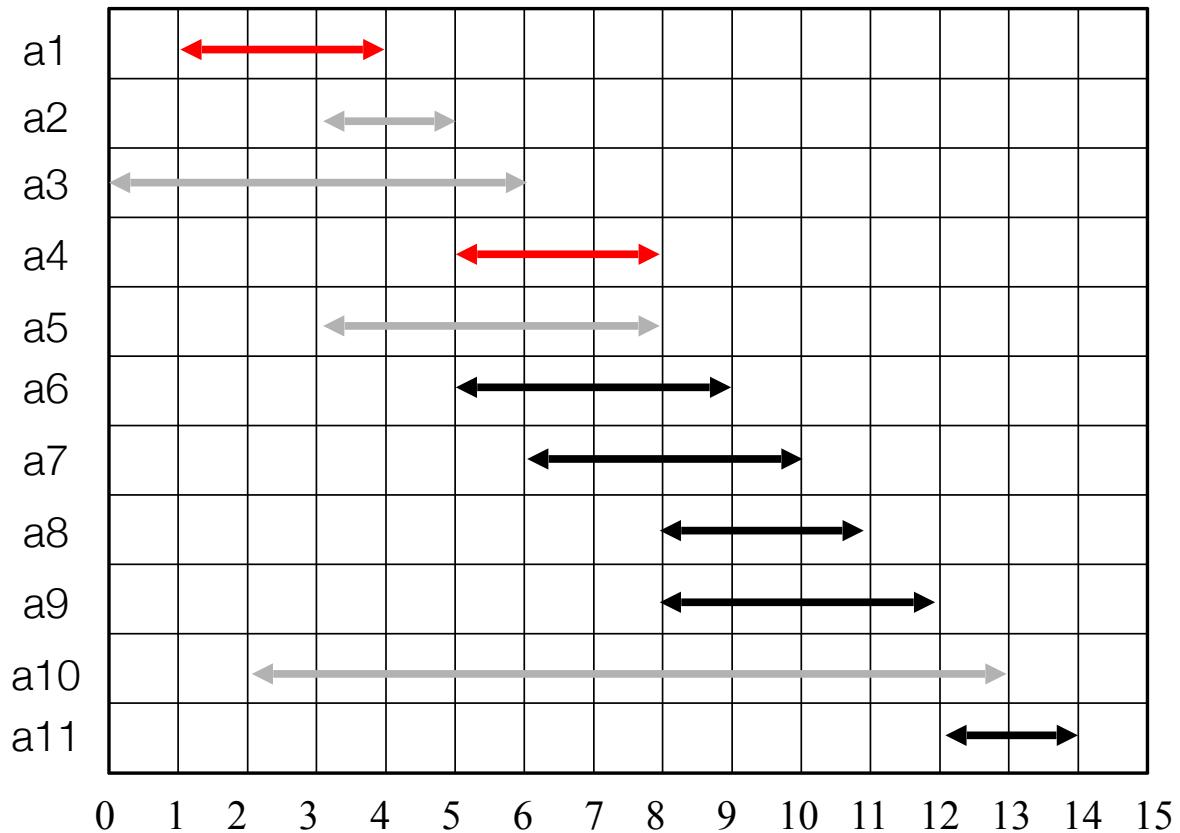


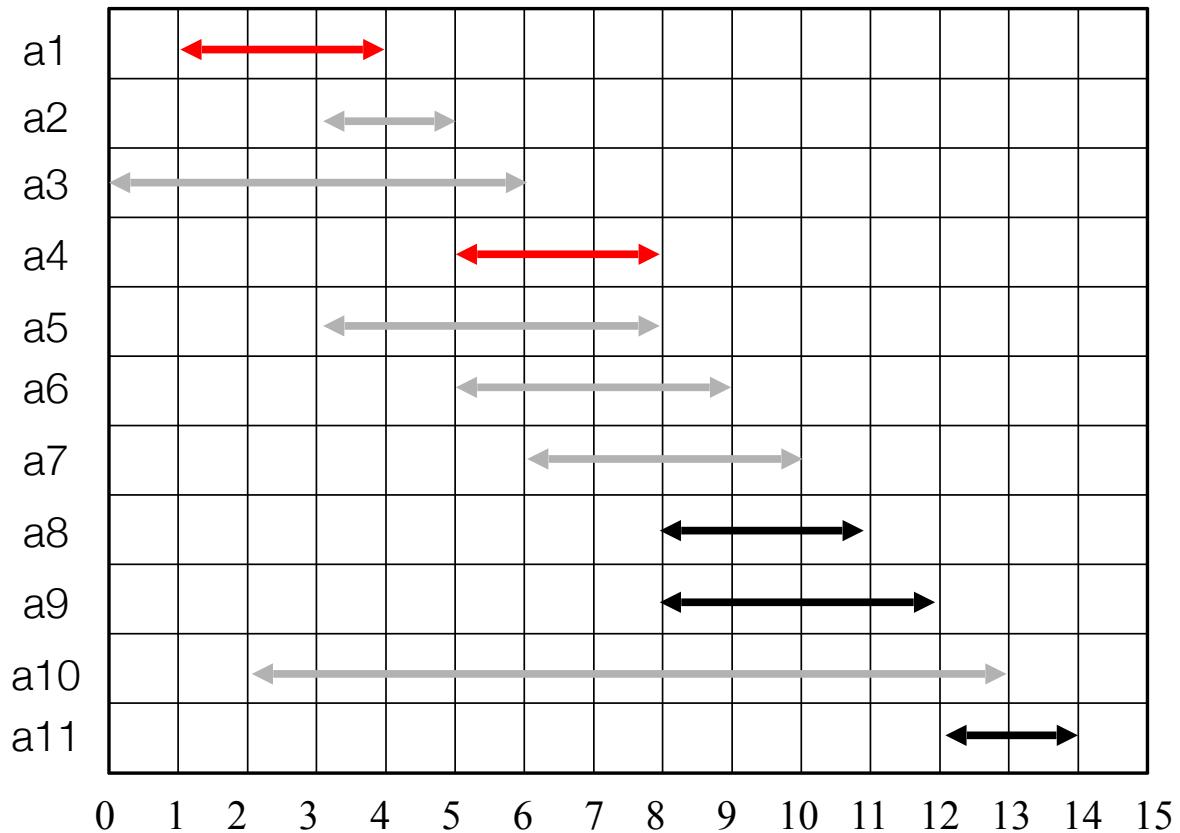
Strategia Greedy: primo che finisce

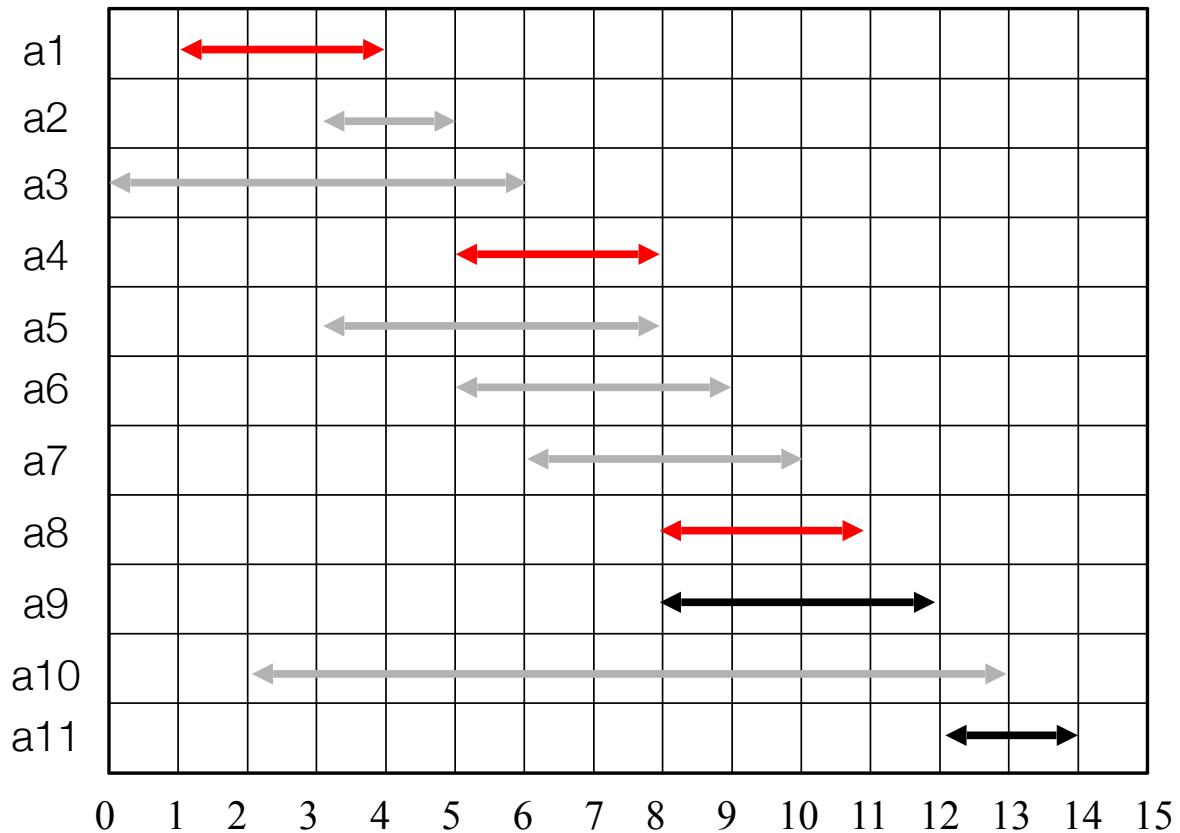
- Seleziona l'attività che finisce prima
- Elimina le attività che non sono compatibili
- Ripeti!

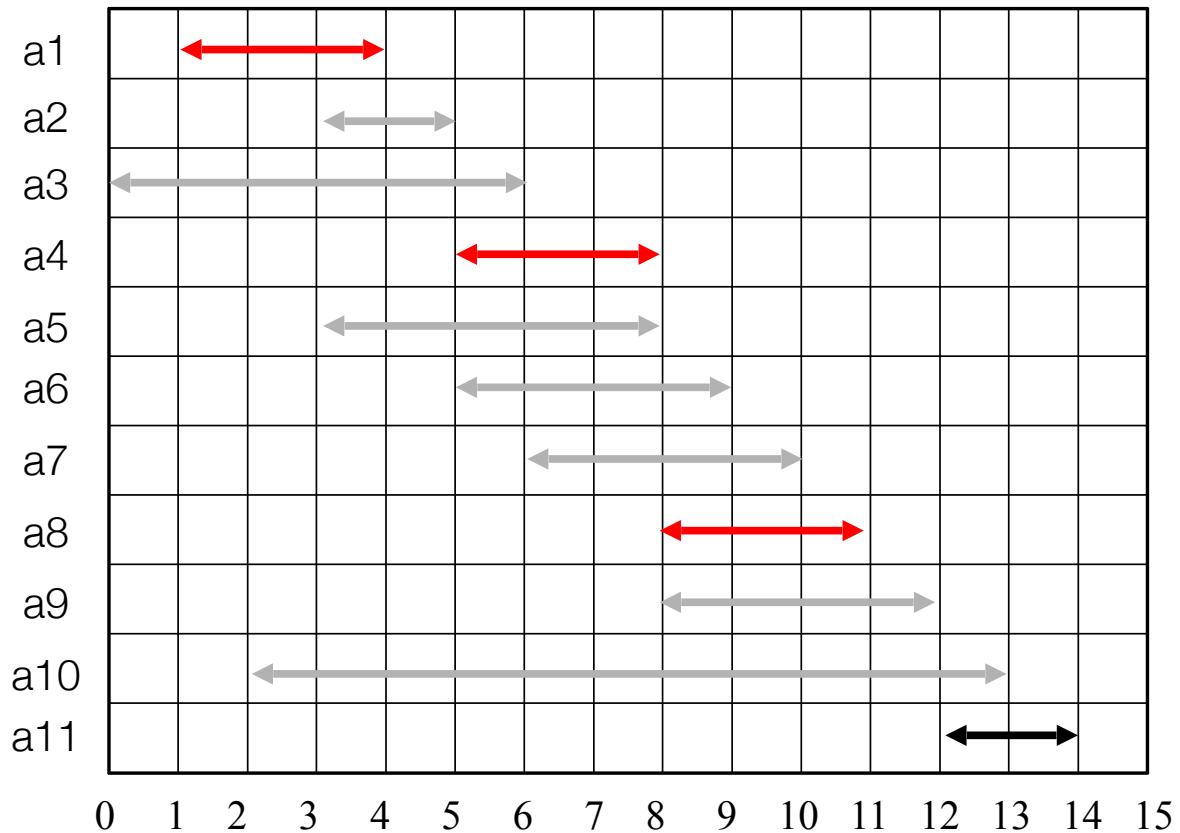


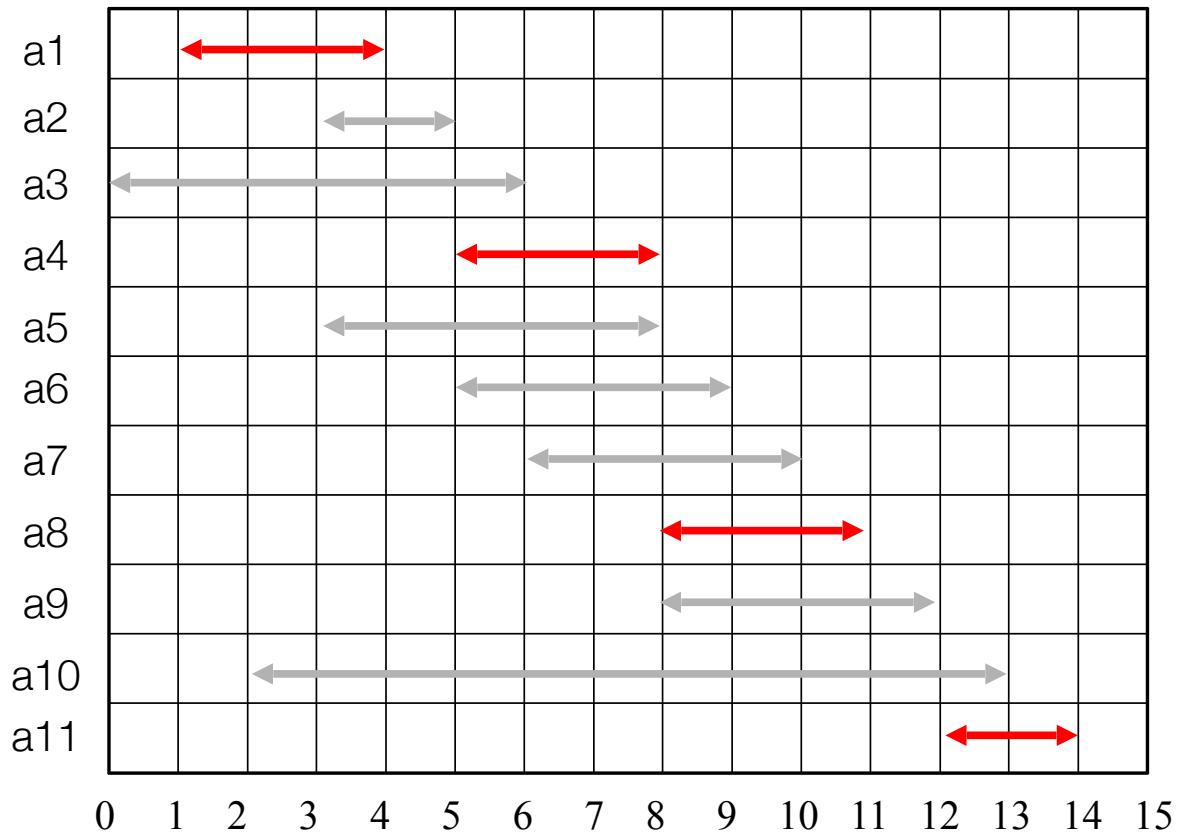












Scheduling di attività – assumendo ordinate per tempo di fine

GREEDY-ACTIVITY-SELECTOR(s, f)

```
1    $n \leftarrow \text{length}[s]$ 
2    $A \leftarrow \{a_1\}$ 
3    $i \leftarrow 1$ 
4   for  $m \leftarrow 2$  to  $n$ 
5       do if  $s_m \geq f_i$ 
6           then  $A \leftarrow A \cup \{a_m\}$ 
7            $i \leftarrow m$ 
8   return  $A$ 
```

Perché Greedy?

- Greedy nel senso che lascia aperte le possibilità di inserimento nella soluzione finale alle attività ancora da schedulare, ove possibile
- La scelta greedy è quella che **massimizza la quantità di tempo rimanente non schedulato**

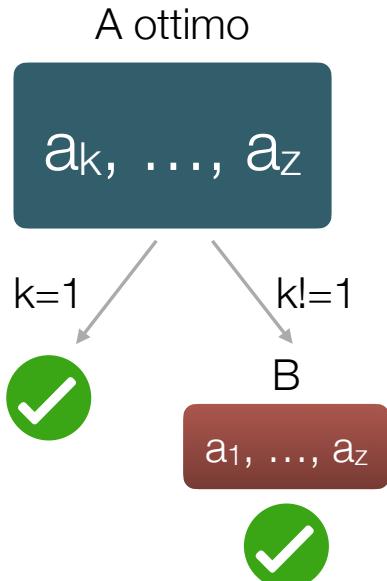
Perché questa soluzione è ottima?

- Per dimostrare che una soluzione greedy è ottima globalmente, bisogna dimostrare che:
 - Il problema ha una sotto-struttura ottima (come per Programmazione Dinamica)
 - La soluzione soddisfa la “greedy-choice property”
 - Dimostrare che ogni scelta greedy conduce all’ottimo globale
 - Soluzione greedy procede top-down, al contrario della Programmazione Dinamica

Greedy-Choice Property

- **Dimostrare** che esiste una soluzione ottima che inizia con la **scelta fatta dalla soluzione greedy** (nel nostro caso con la Attività a_1 , che è sempre presente per costruzione nella soluzione Greedy)
- Supponiamo $A \subseteq S$ sia una soluzione ottima
 - La prima attività in A sia k
 - Se $k = 1$, lo schedule A inizia con la scelta greedy
 - If $k \neq 1$, bisogna dimostrare che esiste una soluzione ottima B per S che inizia con la scelta Greedy, a_1
 - Sia $B = A - \{k\} \cup \{1\}$
 - $f_1 \leq f_k \rightarrow$ attività in B sono ammissibili
 - B ha lo stesso numero di attività di A

Quindi, B è ottimo



Sotto-strutture ottime

- Dopo aver scelto (in maniera greedy) l'attività a_1 , il problema si riduce a quello di trovare una soluzione ottima per il problema generale per quelle attività che restano e che sono compatibili con a_1
- **Sotto-struttura Ottima**
 - Se A è ottimo per S , allora $A' = A - \{a_1\}$ è ottimo per $S' = \{i \in S : s_i \geq f_{I_i}\}$
 - Perché? **Provate a dimostrare per assurdo....**
 - Se potessimo trovare una soluzione B' per S' con più attività di A' , aggiungendo l'attività a_1 a B' avremmo una soluzione B per S più grande di A → **Assurdo!**
- Dopo ogni scelta greedy, restiamo con un **problema di ottimizzazione della stessa forma di quello originale, ma più piccolo**
 - Per induzione sul numero di scelte fatte, fare la **scelta greedy** ad ogni passo **produce una soluzione ottima globale**

Elementi di una strategia Greedy

- Un algoritmo Greedy effettua una serie di scelte, ognuna delle quali è la migliore per il momento in cui è presa
 - **NON** produce sempre una soluzione ottima globale
- Due ingredienti tipicamente propri di problemi che si risolvono bene con strategia Greedy
 - Greedy-choice property
 - Sotto-struttura ottima

Greedy-Choice Property

- Una soluzione ottima globalmente può essere ottenuta effettuando scelte ottime locali (**greedy**)
 - Fai la scelta che sembra migliore al momento, ignorando il futuro, e poi risolvi i sotto-problemi (**top-down**)
- Naturalmente, per dimostrare l'ottimalità globale della soluzione bisogna dimostrare che ogni scelta greedy è parte di una soluzione ottima globale

Sotto-problemi ottimi

- Come visto per la Programmazione Dinamica
- La soluzione ottima per un problema contiene le soluzioni ottime per i sotto-problemi

Come passare l'esame! (esercizio)

Esercizio Programmazione Dinamica

Problema

Come strutturare la tabella (e quindi la formulazione ricorsiva)?

- Si consideri un **esame** con n problemi, ognuno dei quali vale $v[i]$ punti e richiede $t[i]$ minuti per essere risolto
- Il vostro obiettivo è di passare l'esame con il minimo sforzo possibile
- Trovate quindi un algoritmo che dati $v[1 \dots n]$, $t[1 \dots n]$ e un valore V che rappresenta la sufficienza, trova il sottoinsieme di problemi che richiede il minor tempo possibile e permette di passare l'esame

Cose da fare

1. Sia $T(i, v)$ il numero minimo di minuti richiesti per ottenere almeno v risolvendo un qualunque sottoinsieme dei primi i problemi
 - **Scrivere un'espressione ricorsiva** per $T(i, v)$
2. Scrivere **l'algoritmo** basato su programmazione dinamica
3. Valutare la **complessità** dell'algoritmo proposto

Programmazione Dinamica – Casi Base

- Per ottenere almeno 0 o un voto negativo, posso uscire subito dall'aula:
 - $T[i, v] = 0 : \forall i, \forall v \leq 0$
- Se non ho esercizi da risolvere, ma voglio ottenere $v > 0$, resterò chiuso dentro l'aula in eterno:
 - $T[0, v] = +\infty : \forall v > 0$

Programmazione Dinamica – formulazione ricorsiva

- Formulazione ricorsiva del problema può essere espressa come:

$$T[i, v] = \begin{cases} T[i-1, v-v[i]] + t[i] & \text{se risolvo } i \\ & \quad \quad \quad \text{altrimenti} \end{cases}$$

In termini di problemi più piccoli – analizziamo l'esercizio
i-esimo....o lo risolviamo oppure no

Programmazione Dinamica – formulazione ricorsiva

- Formulazione ricorsiva del problema può essere espressa come:

Se $v > 0$, else 0

$$T[i, v] = \begin{cases} T[i-1, v-v[i]] + t[i] & \text{se risolvo } i \\ T[i-1, v] & \text{altrimenti} \end{cases}$$

In termini di problemi più piccoli – analizziamo l'esercizio
i-esimo....o lo risolviamo oppure no

Programmazione Dinamica

```
integer studente-pigro(integer[] v, integer[] t, integer n, integer V)
integer[][] T ← newinteger[0 … n][1 … V]
for v ← 1 to V do
    T[0, v] ← +∞
for i ← 1 to n do
    T[i, v] ← min(T[i - 1, v], t[i] + iif(v - v[i] > 0, M[i - 1, v - v[i]], 0))
return T[n, V]
```

Programmazione Dinamica

```
integer studente-pigro(integer[] v, integer[] t, integer n, integer V)
```

```
    integer[][] T ← newinteger[0...n][1...V]
```

```
    for  $v \leftarrow 1$  to  $V$  do
```

```
         $T[0, v] \leftarrow +\infty$ 
```

```
    for  $i \leftarrow 1$  to  $n$  do
```

```
         $T[i, v] \leftarrow \min(T[i - 1, v], t[i] + \text{iif}(v - v[i] > 0, M[i - 1, v - v[i]], 0))$ 
```

```
    return  $T[n, V]$ 
```

Complessità? $O(nV)$

Pseudo-polinomiale

Infine....



Supponete ora che risolvere un esercizio in **maniera parziale** dia la possibilità di ottenere una **porzione di voto proporzionale**. Descrivete un algoritmo di costo $O(n \log n)$ per risolvere lo stesso problema....

Un **approccio greedy** può funzionare come con lo zaino frazionario

- **Ordinare** i problemi per $m[i]/v[i]$ crescente (o per $v[i]/m[i]$ decrescente)
- A questo punto, **iteriamo sulla lista ordinata**, fino a quando V non è stato raggiunto
- Solo l'ultimo problema può essere risolto parzialmente

Algoritmi pseudo-polinomiali

- **Test:** quali dei seguenti algoritmi sono **pseudo-polinomiali** e quali no??
 - Sequenza ottima per la moltiplicazione di matrici
 - NO
 - LCS
 - NO
 - Partizione di n interi di somma totale 2^s
 - SI
 - Zaino 0-1
 - SI
 - Come passare l'esame
 - SI

Ci vediamo all'esame....

Buon lavoro a tutti e buone vacanze!!!