ALGORITMI E STRUTTURE DATI

Prof. Manuela Montangero

A.A. 2022/23

Programmazione Dinamica

Shortest Paths su DAG - Longest Increasing Subsequence

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Problemi di ottimizzazione

Alcune tecniche per affrontare problemi di ottimizzazione

Algoritmi GREEDY (Golosi o Ingordi):
costituisce la soluzione in maniera incrementale, effettuando una serie di scelte in cui, ad ogni passo, si seleziona una parte della soluzione che permette di ottimizzare il costo della soluzione parziale costruita fino a quel momento.

Algoritmi di PROGRAMMAZIONE DINAMICA:
(se non si sa quali sottoproblemi risolvere) risolvere TUTTI i sottoproblemi e conservare i risultati ottenuti per utilizzarli successivamente (se servono) nel risolvere il problema di dimensione più grande.

Programmazione Dinamica

La **PROGRAMMAZIONE DINAMICA** può essere applicata se il **PROBLEMA** ha una **SOTTOSTUTTURA OTTIMA**:

Programmazione non nel senso di **coding**, ma di **pianificare**. (Bellman 1950s)

- È possibile combinare le soluzioni ottime dei sottoproblemi per trovare la soluzione di un problema più grande
- Le scelte fatte per risolvere i sottoproblemi in modo ottimo non devono essere modificate quando la soluzione al sottoproblema diventa una parte della soluzione al problema più grande

Esempio: se un nodo v si trova sul cammino minimo P da s a u,
allora la parte del cammino P da s a v è minimo per v

Programmazione Dinamica

La **PROGRAMMAZIONE DINAMICA**
può essere applicata se il **PROBLEMA** ha una
SOTTOSTUTTURA OTTIMA:

- È possibile combinare le soluzioni ottime dei sottoproblemi per trovare la soluzione di un problema più grande
- Le scelte fatte per risolvere i sottoproblemi in modo ottimo non devono essere modificate quando la soluzione al sottoproblema diventa una parte della soluzione al problema più grande

Affinché l'algoritmo abbia **costo computazionale polinomiale** occorre che:

- il numero di sottoproblemi da risolvere sia polinomiale
- il tempo per combinare soluzioni dei sottoproblemi sia polinomiale

Si memorizzano le soluzioni dei sottoproblemi in una matrice,
senza sapere quali saranno necessarie e quali no

Cammini minimi su DAG

Usiamo la programmazione dinamica

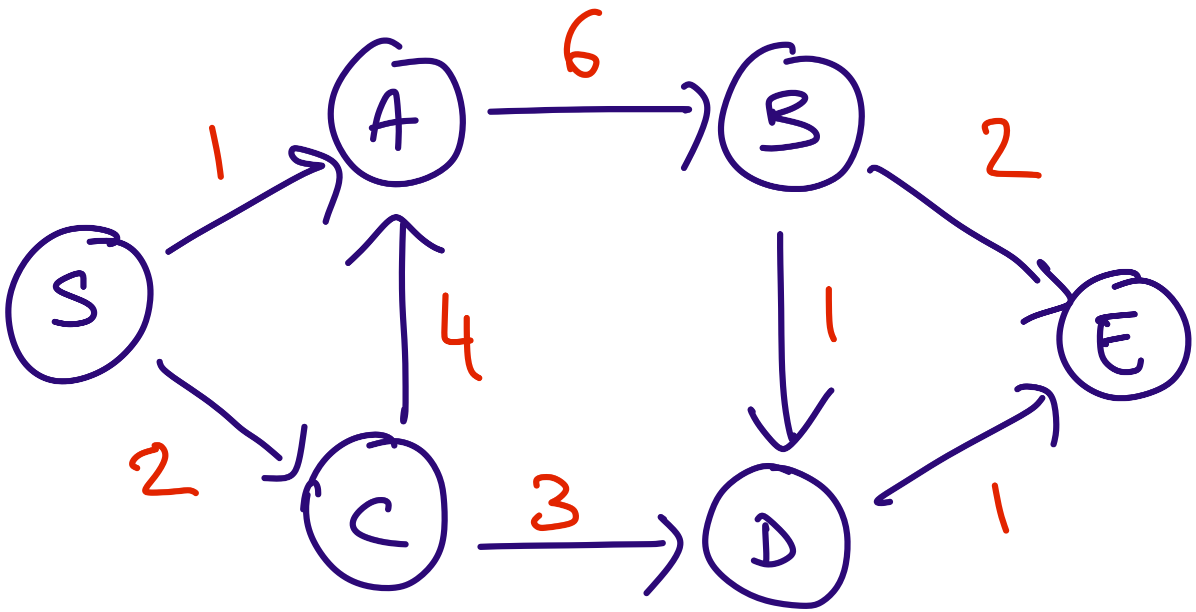
PROBLEMA:

INPUT: DAG G=(V,E),

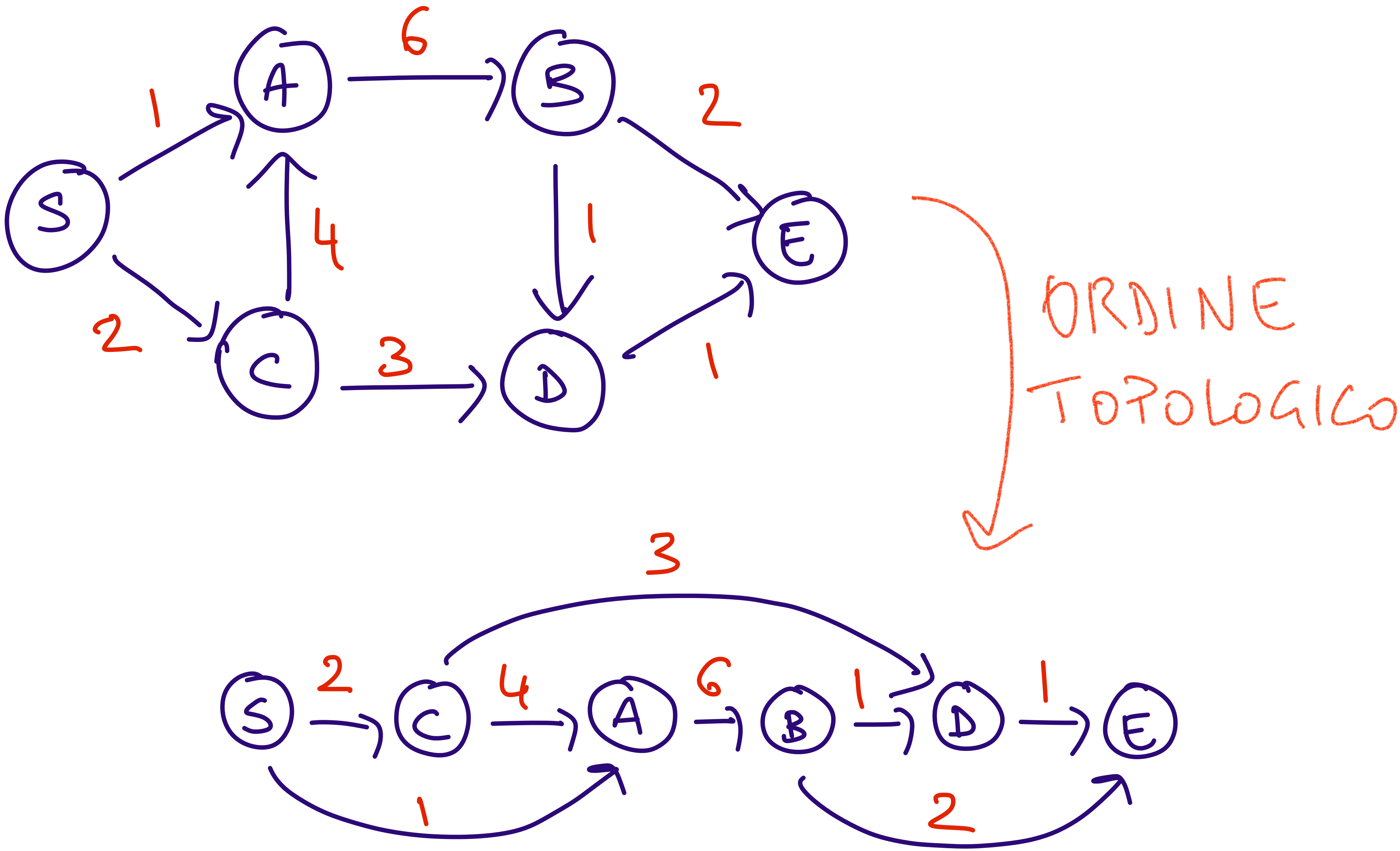
un nodo sorgente s di G

funzione costo sugli archi c: E—>R

OUTPUT: **distanze minime** dei nodi di G da s

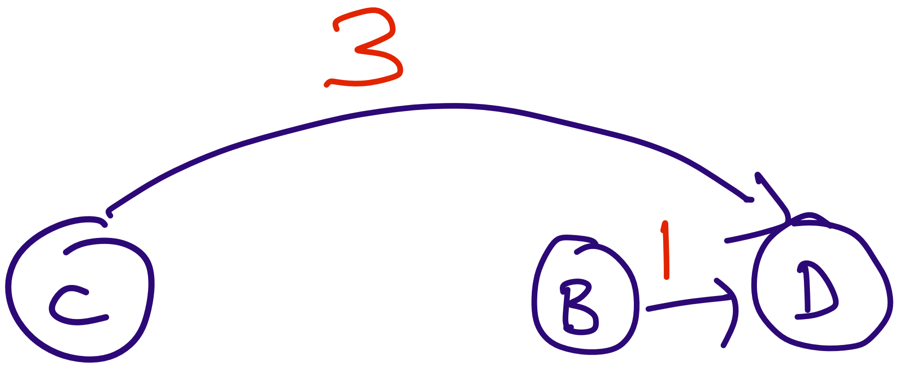


Cammini minimi su DAG



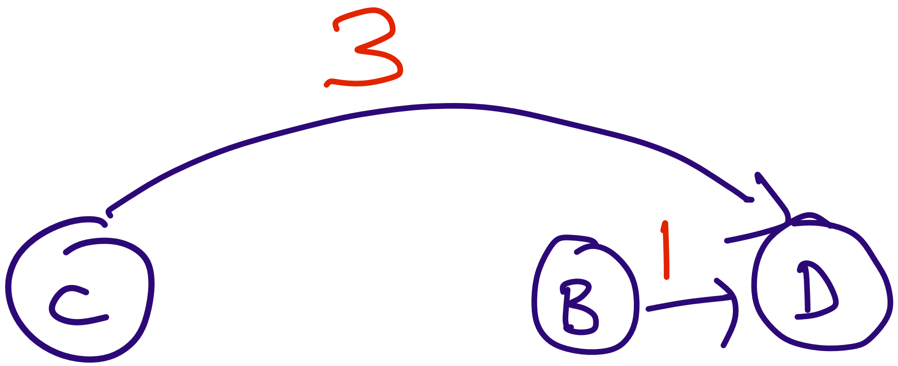
 UNIMORE

Cammini minimi su DAG



Il nodo D è raggiungibile solo dai nodi C e B

Cammini minimi su DAG



Il nodo D è raggiungibile solo dai nodi C e B

$$\texttt{dist[B]} = \min \begin{cases} \texttt{dist[C]} + 3 \\ \texttt{dist[D]} + 1 \end{cases}$$

Se calcolo prima la distanza minima di C e B dalla sorgente, posso calcolare facilmente la distanza minima di D dalla sorgente

Cammini minimi su DAG

SOTTOPROBLEMA (relativo al nodo v):
dist[v] distanza minima del nodo v dalla sorgente

Numero di sottoproblemi:
uno per ogni nodo del DAG —> |V|

Comporre sottoproblemi:
il sottoproblema relativo ad un nodo v si risolve componendo i sottoproblemi relativi ai nodi che hanno un arco diretto verso v, con semplici operazioni aritmetiche

Ordine in cui risolvere i sottoproblemi:
ordine topologico nodi del DAG

Cammini minimi su DAG

```
Shortes_Path_DAG(G=(V,E),c,s)
TS := Topological_sort(G)
for all v ∈ V do
   dist[v] := +∞
dist[s] := 0
v := pop(TS)
while v ≠ s do
   v := pop(TS)
while NOT is_empty(TS) do
   dist[v] :=  min {dist[u] + c(u,v)}
             (u,v)∈E
   v := pop(TS)
return dist[]
```

Restituisce una PILA

I nodi che precedono s nell'ordinamento topologico NON sono raggiungibili da s (e s sta sicuramente nella PILA)
Alla fine del **while** abbiamo che v = s

Se c'è un arco (u,v) tale che u precede s nell'ordinamento topologico, allora dist[u] sarà uguale a più infinto e u non verrà mai scelto se esiste almeno un arco (u',v) tale che u' segue s nell'ordinamento topologico.
Se tale arco non esiste, allora dist[v] rimane più infinto ad indicare che non esiste un cammino da s a v

Cammini minimi su DAG

```
Shortes_Path_DAG(G=(V,E),c,s)
TS := Topological_sort(G)
for all v ∈ V do
   dist[v] := +∞
dist[s] := 0
v := pop(TS)
while v ≠ s do
   v := pop(TS)
while NOT is_empty(TS) do
   dist[v] :=  min     {dist[u] + c(u,v)}
             (u,v)∈E
   v := pop(TS)
return dist[]
```

$O(|V|)$

$O(|V|)$

tutte le ripetizioni $O(|E|)$

tutte le ripetizioni $O(|V|)$

Costo computazionale $O(|V|+|E|)$

Cammini minimi su DAG

Come abbiamo proceduto?

1) Abbiamo risolto sottoproblemi

abbiamo calcolato `dist[v]` per ogni $v \in V$

2) Abbiamo risolto prima sottoproblemi più piccoli

per ogni `v`, avevamo già risolto i problemi dei nodi più vicini ad `s` che si trovavano sui cammini da s a v
(avevamo già tutti i `dist[u]` per ogni $(u,v) \in E$)

3) Abbiamo composto il risultato di sottoproblemi più piccoli, per trovare soluzioni a problemi, più grandi con semplici operazioni aritmetiche

$$\mathtt{dist[v]} := \min_{(u,v)\in E} \{\mathtt{dist[u]} + \mathtt{c(u,v)}\}$$

4) Abbiamo seguito un ordine ben preciso per risolvere i sottoproblemi

L'ordine è dato dall'ordine topologico

Longest Increasing Subsequence (LIS)

Sottosequenza crescente più lunga

PROBLEMA:

INPUT: Sequenza di n numeri $A = < a_1, a_2, \ldots, a_n >$

OUTPUT: la più lunga sottosequenza crescente in A

Definizione (sottosequenza):

Data una sequenza di elementi $A = < a_1, a_2, \ldots, a_n >$ e dato un insieme di $k \in [1..n]$ indici $I = \{i_1, i_2, \ldots, i_k\} \subseteq \{1,2,\ldots,n\}$ tale che $1 \leq i_1 < i_2 < \ldots < i_k \leq n$, la sequenza $A' = < a_{i_1}, a_{i_2}, \ldots, a_{i_k} >$ è una sottosequenza di A

$$A = < 5,2,8,6,3,9,7 > \qquad A' = < 2,8,7 >$$

Definizione (sottosequenza crescente):

Data una sequenza $A = < a_1, a_2, \ldots, a_n >$ e una sua sottosequenza $A' = < a_{i_1}, a_{i_2}, \ldots, a_{i_k} >$, A' è crescente se e solo se $a_{i_1} < a_{i_2} < \ldots < a_{i_k}$

$$A = < 5,2,8,6,3,9,7 > \qquad A' = < 2,8,9 >$$

Longest Increasing Subsequence (LIS)

PROBLEMA:

INPUT: Sequenza di n numeri $A = < a_1, a_2, \ldots, a_n >$

OUTPUT: la più lunga sottosequenza crescente in A

soluzione ammissibile: sottosequenza crescente

costo soluzione: numero di valori nella sottosequenza

funzione obiettivo: massimo

soluzione ottima: sottosequenza crescente più lunga

Longest Increasing Subsequence (LIS)

PROBLEMA:

INPUT: Sequenza di n numeri $A = < a_1, a_2, \ldots, a_n >$

OUTPUT: la più lunga sottosequenza crescente in A

$$A = < 5,2,8,6,3,6,9,7 >$$

FORZA BRUTA: per ogni sottosequenza, controllare se è crescente oppure no. Tenere traccia della sottosequenza più lunga trovata.

Longest Increasing Subsequence (LIS)

PROBLEMA:

INPUT: Sequenza di n numeri $A = < a_1, a_2, \ldots, a_n >$

OUTPUT: la più lunga sottosequenza crescente in A

$$A = < 5,2,8,6,3,6,9,7 >$$

NO

FORZA BRUTA: per ogni sottosequenza, controllare se è crescente oppure no. Tenere traccia della sottosequenza più lunga trovata.

Longest Increasing Subsequence (LSI)

PROBLEMA:

INPUT: Sequenza di n numeri $A = < a_1, a_2, \ldots, a_n >$

OUTPUT: la più lunga sottosequenza crescente in A

SI

$$A = < 5,2,8,6,3,6,9,7 >$$

FORZA BRUTA: per ogni sottosequenza, controllare se è crescente oppure no. Tenere traccia della sottosequenza più lunga trovata.

Longest Increasing Subsequence (LSI)

PROBLEMA:

INPUT: Sequenza di n numeri $A = < a_1, a_2, \ldots, a_n >$

OUTPUT: la più lunga sottosequenza crescente in A

$$A = < 5,2,8,6,3,6,9,7 >$$

FORZA BRUTA: per ogni sottosequenza, controllare se è crescente oppure no. Tenere traccia della sottosequenza più lunga trovata.

Costo computazionale?

Dipende dal numero di sottoinsiemi degli indici $\Rightarrow O(2^n)$

Longest Increasing Subsequence (LSI)

PROBLEMA:

INPUT: Sequenza di n numeri $A = < a_1, a_2, \ldots, a_n >$

OUTPUT: la più lunga sottosequenza crescente in A

$$A = < 5,2,8,6,3,\boxed{6},9,7 >$$

ALTRA IDEA: considero un numero alla volta e provo a vedere se si può allungare una sequenza crescente che si trova alla sinistra del numero

Longest Increasing Subsequence (LSI)

PROBLEMA:

INPUT: Sequenza di n numeri $A = < a_1, a_2, \ldots, a_n >$

OUTPUT: la più lunga sottosequenza crescente in A

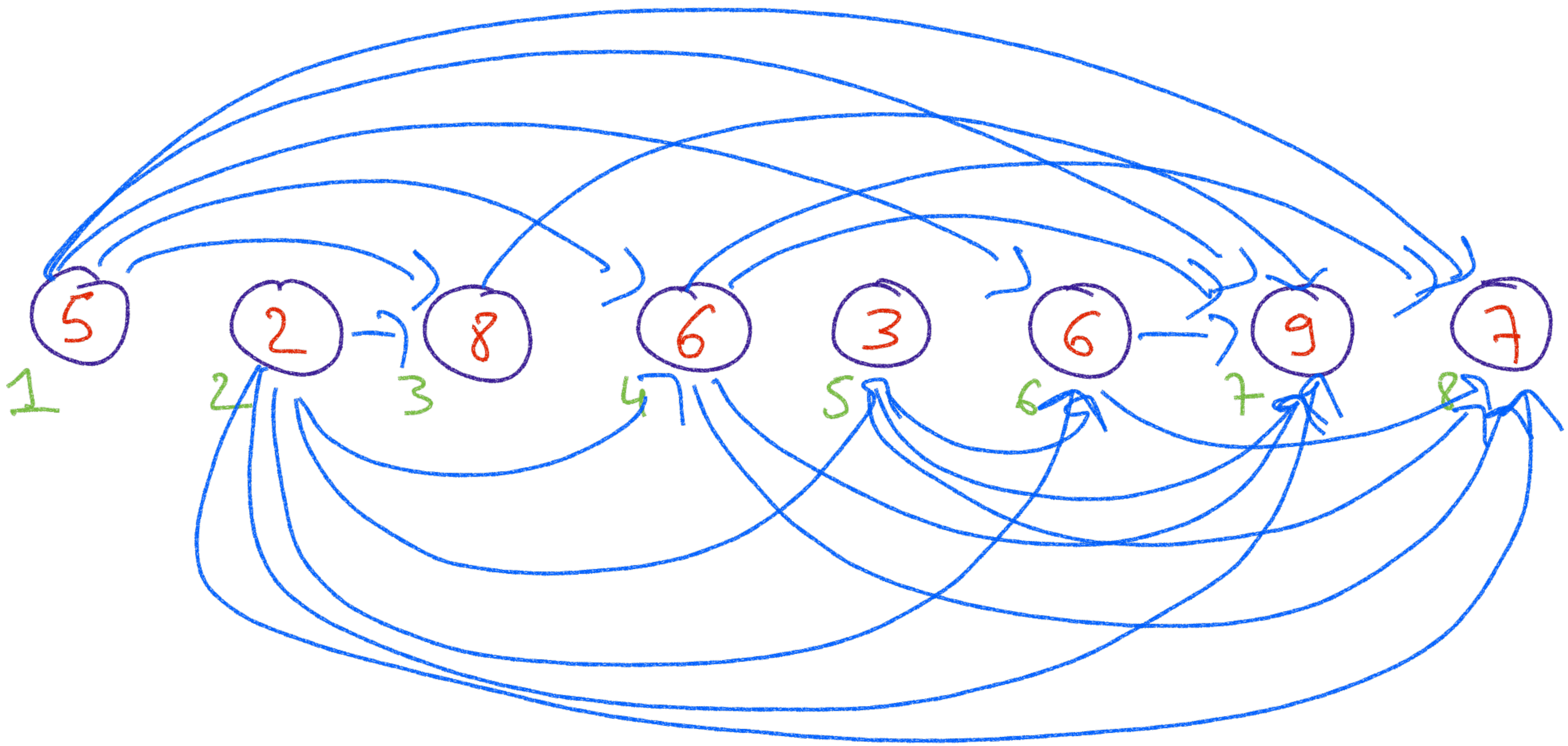
$$A = < 5,2,8,6,\boxed{3},6,9,7 >$$

ALTRA IDEA: considero un numero alla volta e provo a vedere se si può allungare una sequenza crescente che si trova alla sinistra del numero

Longest Increasing Subsequence (LSI)

Costruiamo un DAG in cui:

- Esiste un nodo per ogni elemento della sequenza
- I nodi sono ordinati linearmente secondo l'ordine in cui compaiono nella sequenza
- Esiste un arco tra due nodi i e j che corrispondono a a_i e a_j se e solo se $a_i < a_j$

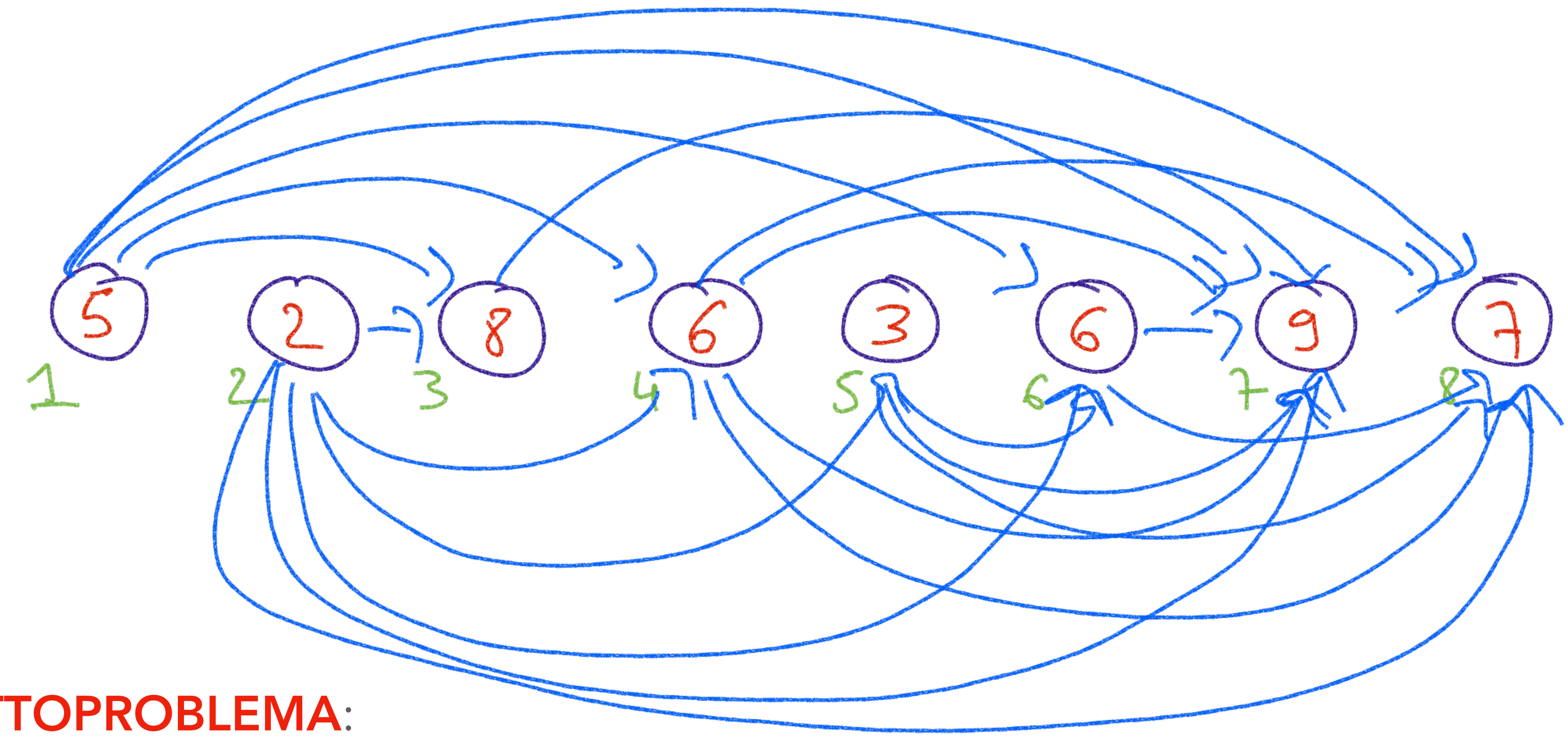


Il problema equivale a trovare il cammino più lungo nel DAG

Longest Increasing Subsequence (LSI)

Costruiamo un DAG in cui:

- Esiste un nodo per ogni elemento della sequenza
- I nodi sono ordinati linearmente secondo l'ordine in cui compaiono nella sequenza
- Esiste un arco tra due nodi i e j che corrispondono a a_i e a_j se e solo se $a_i < a_j$



SOTTOPROBLEMA:

$L[j]$ = lunghezza del cammino più lungo che termina nel nodo j

Longest Increasing Subsequence (LSI)

Costruiamo un DAG in cui:

- Esiste un nodo per ogni elemento della sequenza
- I nodi sono ordinati linearmente secondo l'ordine in cui compaiono nella sequenza
- Esiste un arco tra due nodi i e j che corrispondono a a_i e a_j se e solo se $a_i < a_j$

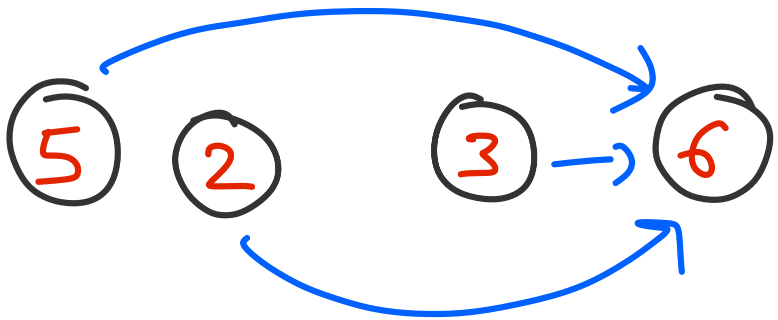
SOTTOPROBLEMA:

$L[j]$ = lunghezza del cammino più lungo che termina nel nodo j

Possiamo calcolare $L[j]$ in funzione
delle soluzioni di sottoproblemi più piccoli,
ovvero di $L[i]$ con $i < j$

$$L[j] = \max_{(i,j)\in E} \{L[i]\} + 1$$

Soluzione ottima del sottoproblema



Longest Increasing Subsequence (LSI)

SOTTOPROBLEMA:

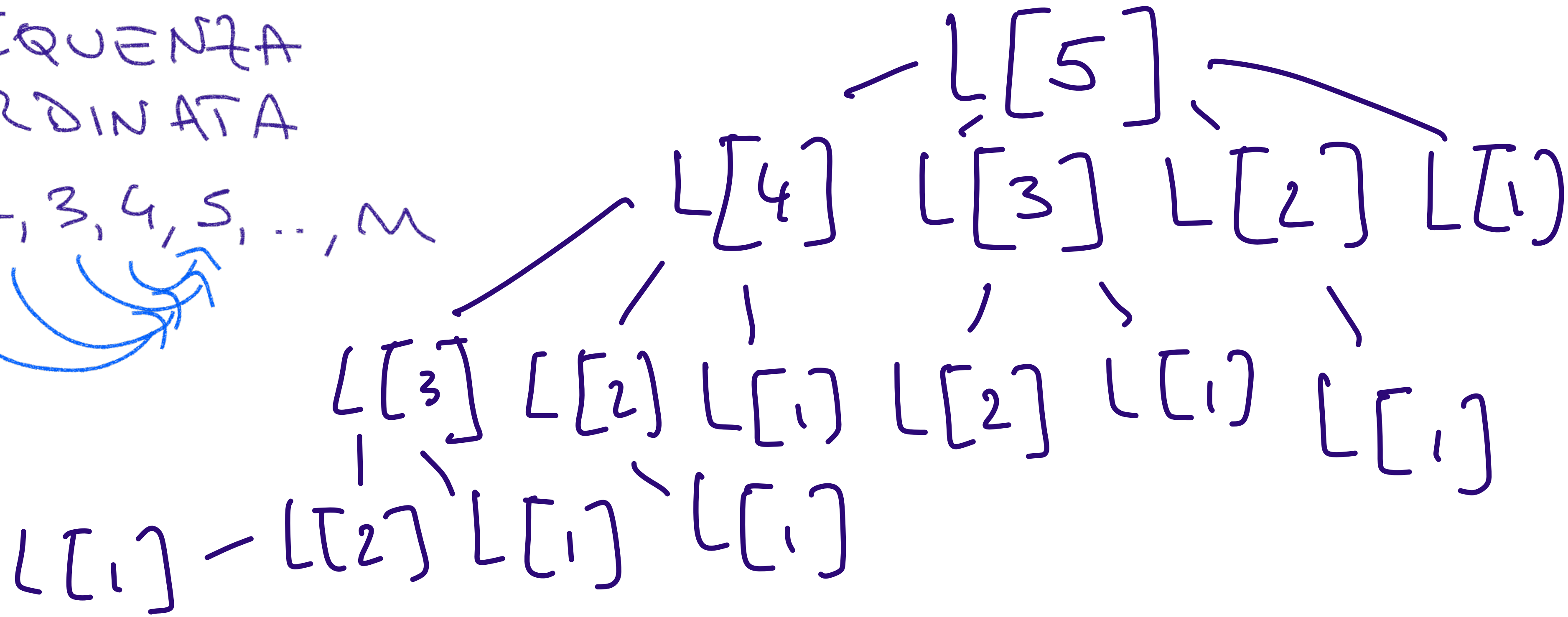
$L[j]$ = lunghezza del cammino più lungo che termina nel nodo j

$$L[j] = \max_{(i,j)\in E} \{L[i]\} + 1$$

Per calcolare $L[j]$ possiamo usare la RICORSIONE:

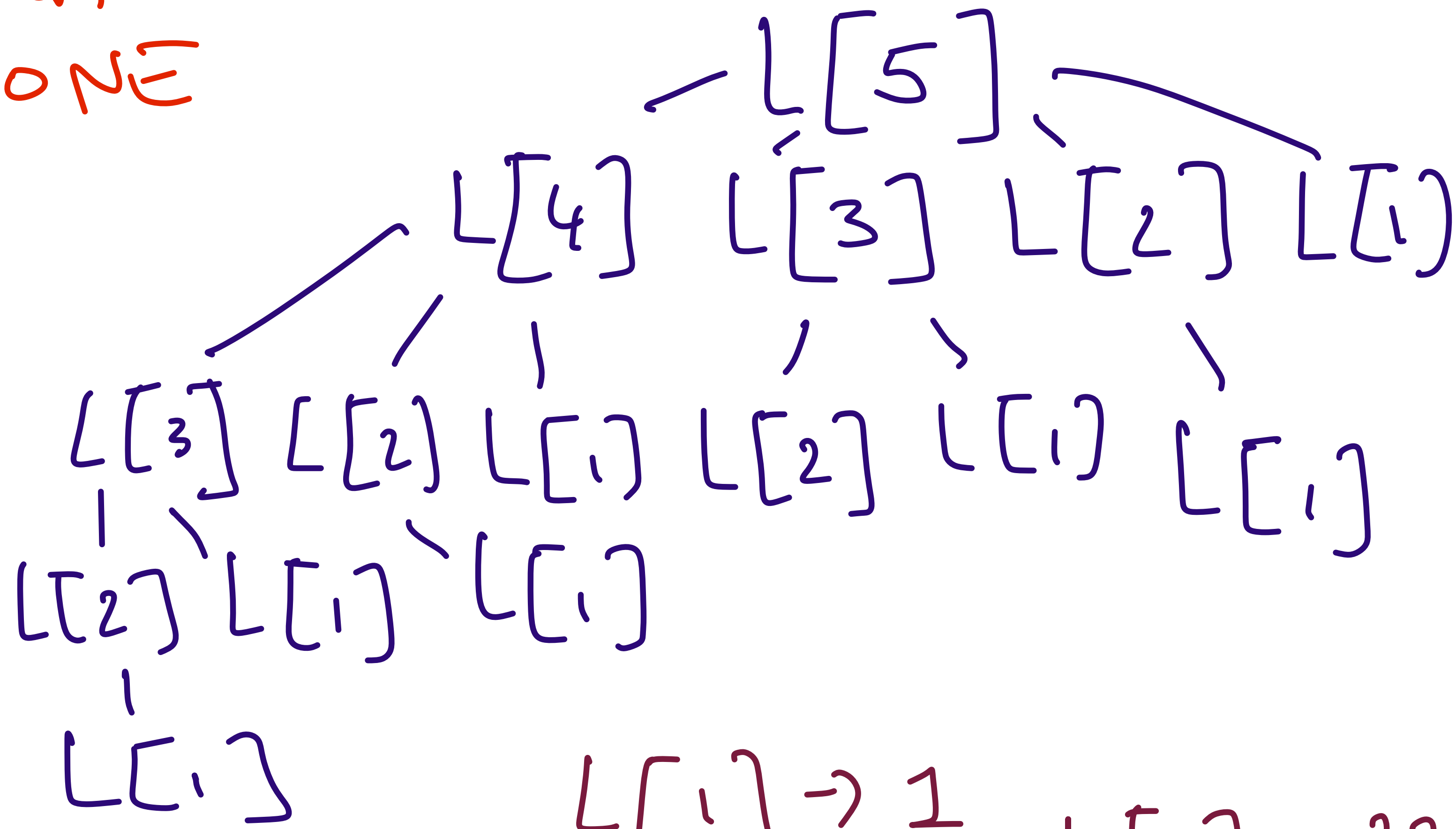
SEQUENZA ORDINATA

1, 2, 3, 4, 5, ..., n



UNIMORE

USIAMO LA RICORSIONE

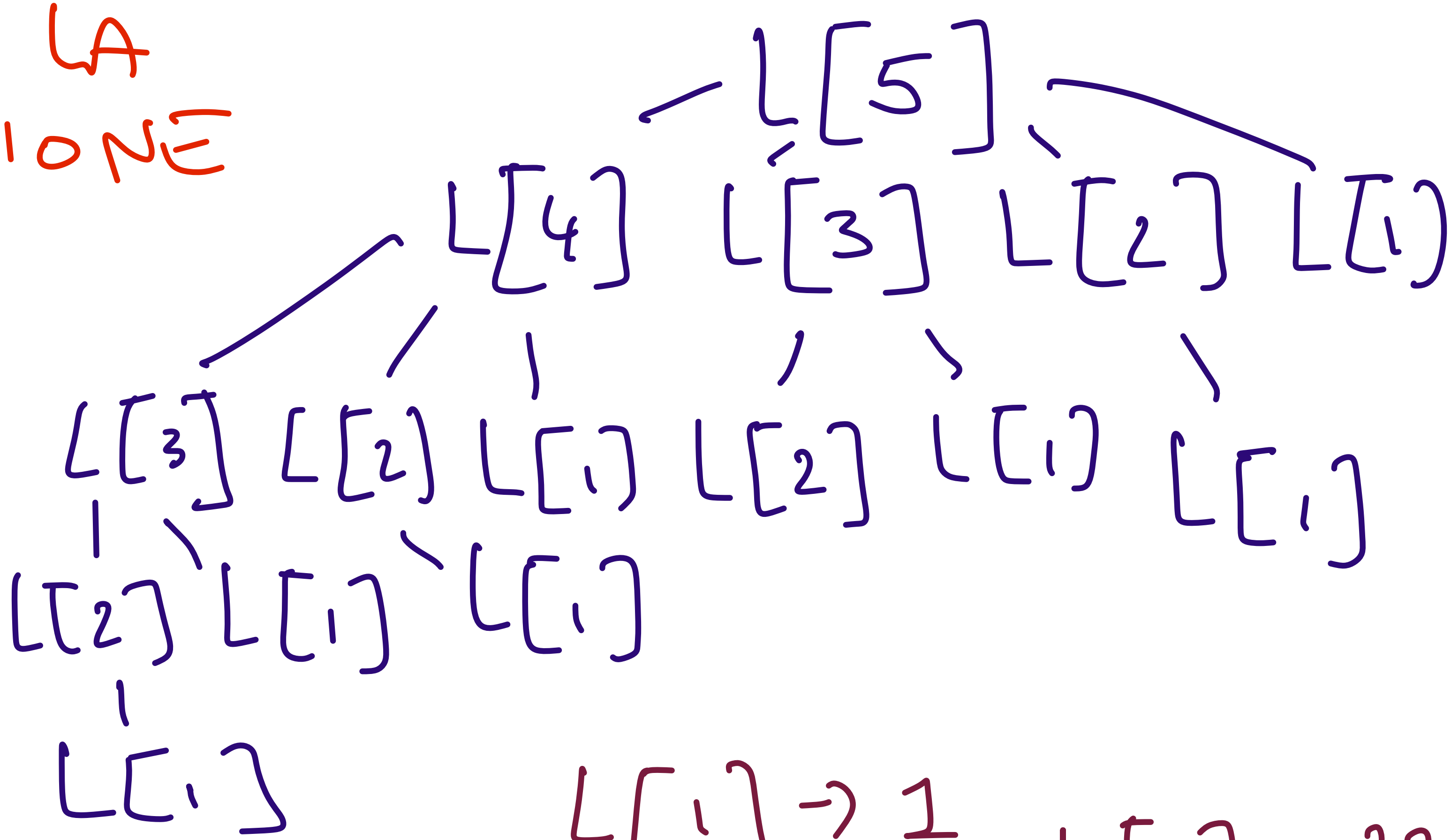


$L[1] \to 1$
$L[2] \to 2$
$L[3] \to 4$
$L[4] \to 8$
$L[5] \to 16$

$L[6] \to$??

QUANTE CHIAMATE RICORSIVE?

USIAMO LA RICORSIONE



QUANTE CHIAMATE RICORSIVE?

$L[1] \rightarrow 1$

$L[2] \rightarrow 2$

$L[3] \rightarrow 4$

$L[4] \rightarrow 8$

$L[5] \rightarrow 16$

$L[6] \rightarrow$??

$L[k] \rightarrow 2^{k-1}$

VOGLIAMO DIMOSTRARE CHE

PER CALCOLARE $L[k]$
SONO NECESSARIE $f(k) = 2^{k-1}$
CHIAMATE RICORSIVE

DIMOSTRAZIONE PER INDUZIONE

CASO BASE: $k=1$, $L[1] \rightarrow 1 = 2^0 = 2^{k-1}$

IPOTESI INDUTTIVA:
PER $J = 1, \ldots, k-1$, SONO NECESSARIE
$f(J) = 2^{J-1}$ CHIAMATE RICORSIVE

PASSO INDUTTIVO!

$L[k]$

$L[k-1]$ $L[k-2]$ ... $L[1]$

$$f(k) = 1 + \sum_{j=1}^{k-1} f(j) =$$

$$= 1 + \sum_{j=1}^{k-1} 2^{j-1} = 1 + \sum_{j=0}^{k-2} 2^{j} = 1 + \left(2^{k-1} - 1\right) =$$

$$= 2^{k-1}$$

somma di potenze di 2

Longest Increasing Subsequence (LSI)

SOTTOPROBLEMA:

$L[j]$ = lunghezza del cammino più lungo che termina nel nodo j

$$L[j] = \max_{(i,j) \in E} \{L[i]\} + 1$$

Per calcolare $L[j]$ possiamo usare la RICORSIONE:

NON E' UNA BUONA IDEA!

COSTO COMPUTAZIONALE ESPONENZIALE!

Longest Increasing Subsequence (LSI)

SOTTOPROBLEMA:

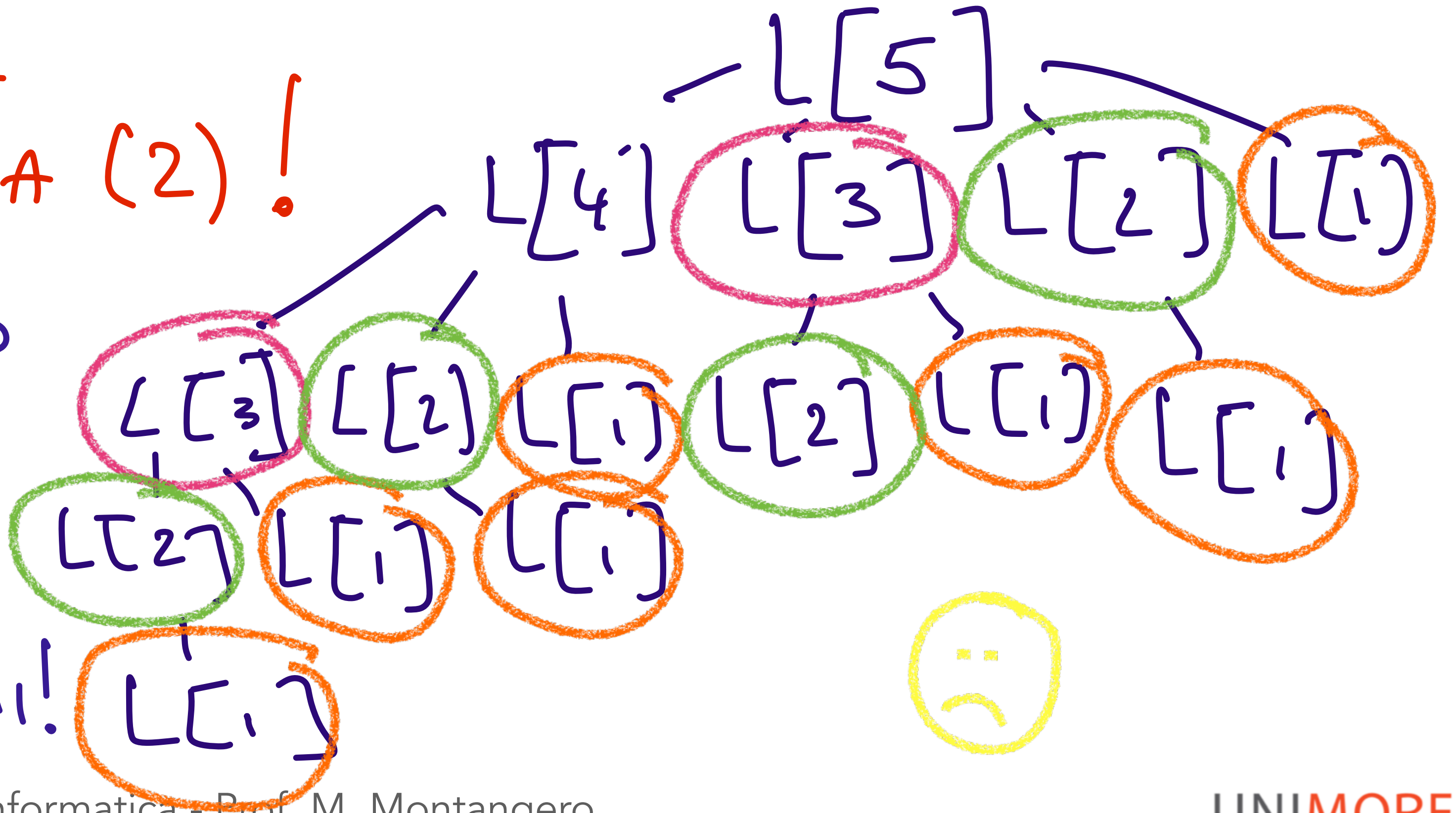
$L[j]$ = lunghezza del cammino più lungo che termina nel nodo j

$$L[j] = \max_{(i,j)\in E} \{L[i]\} + 1$$

Per calcolare $L[j]$ possiamo usare la RICORSIONE:

NON E' UNA BUONA IDEA (2)!

SI RISOLVONO PIU' VOLTE GLI STESSI SOTTOPROBLEMI!



Memoization (memoizzazione)

SALVIAMO I VALORI DEI SOTTOPRO= BLEMI CHE ABBIAMO GIÀ CALCOLATO IN MODO DA NON DOVERLI CALCOLARE NUOVAMENTE

L[5]

L[4] L[3] L[2] L[1]

GIÀ CALCOLATI

BASTA CALCOLARE IL MAX

Memoization (memoizzazione)

SALVIAMO I VALORI DEI SOTTOPRO-
BLEMI CHE ABBIAMO GIÀ CALCOLATO
IN MODO DA NON DOVERLI CALCOLARE
NUOVAMENTE

- IN PRATICA, C'È L'OVERHEAD
DELLA RICORSIONE

Longest Increasing Subsequence (LSI)

SOTTOPROBLEMA:

$L[j]$ = lunghezza del cammino più lungo che termina nel nodo j

$$L[j] = \max_{(i,j)\in E} \{L[i]\} + 1$$

Per calcolare tutti gli $L[j]$ possiamo usare l'ITERAZIONE:

```
for j = 1 to n do
```
$$L[j] = \max_{(i,j)\in E} \{L[i]\} + 1$$

Poiché il grafo è un DAG, gli archi sono diretti dal nodo in posizione i al nodo i posizione j tale che $i < j$

quindi per ogni j è possibile calcolare $L[i]$

Longest Increasing Subsequence (LSI)

SOTTOPROBLEMA:

$L[j]$ = lunghezza del cammino più lungo che termina nel nodo j

$$L[j] = \max_{(i,j)\in E} \{L[i]\} + 1$$

Per calcolare tutti gli $L[j]$ possiamo usare l'ITERAZIONE:

```
for j = 1 to n do
```
$$L[j] = \max_{(i,j)\in E} \{L[i]\} + 1$$

E qual è la soluzione al problema?

Il massimo tra tutti gli $L[j]$ che abbiamo calcolato!

Longest Increasing Subsequence (LSI)

```
LSI(A)
costruire il DAG G a partire da A
for j = 1 to n do
 L[j] := 1
for j = 2 to n do
```
$$L[j] = \max_{(i,j)\in E} \{L[i]\} + 1$$
```
return max_j L[j]
```

Esempio

$A = (5, 2, 8, 6, 3, 6, 9, 7)$

	1	2	3	4	5	6	7	8
L	1							

Esempio

$A = (5, 2, 8, 6, 3, 6, 9, 7)$

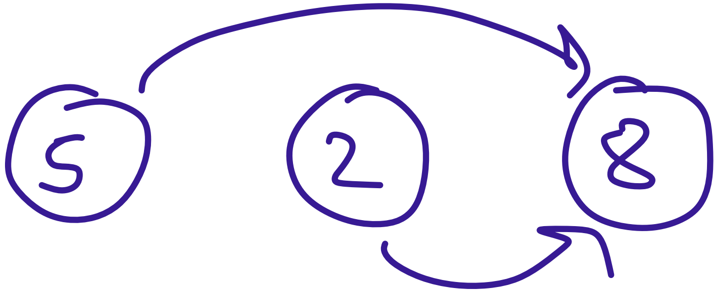
	1	2	3	4	5	6	7	8
L	1	1						

2

Esempio

$A = \langle 5, 2, 8, 6, 3, 6, 9, 7 \rangle$

	1	2	3	4	5	6	7	8
L	1	1	2					

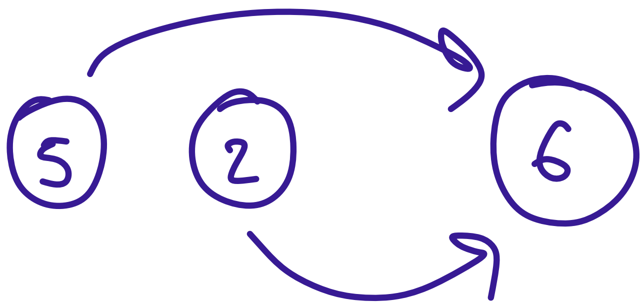


$$L[3] = \max\{L[2], L[1]\} + 1 = $$
$$= \max\{1, 1\} + 1 = $$
$$2$$

Esempio

$A = (5, 2, 8, 6, 3, 6, 9, 7)$

	1	2	3	4	5	6	7	8
L	1	1	2	2				



$$L[4] = \max\{L[2], L[1]\} + 1 =$$

$$= \max\{1, 1\} + 1 =$$

$$= 2$$

Esempio

$$A = (5, 2, 8, 6, 3, 6, 9, 7)$$

	1	2	3	4	5	6	7	8
L	1	1	2	2	2			

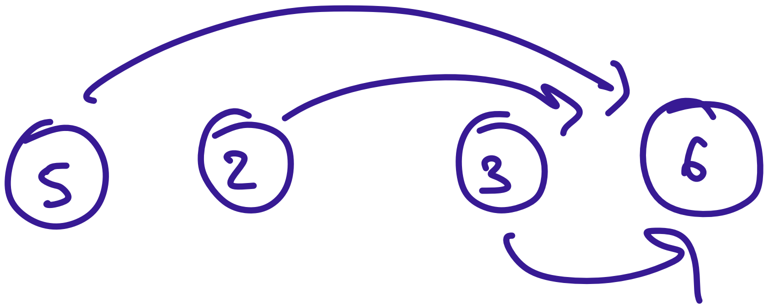
$$2 \rightarrow 3$$

$$L[5] = \max\{L[2]\}+1 = \max\{1\}+1 = 2$$

Esempio

$A = (5, 2, 8, 6, 3, 6, 9, 7)$

	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3		



$L[6] = \max\{L[5], L[2], L[1]\}+1$

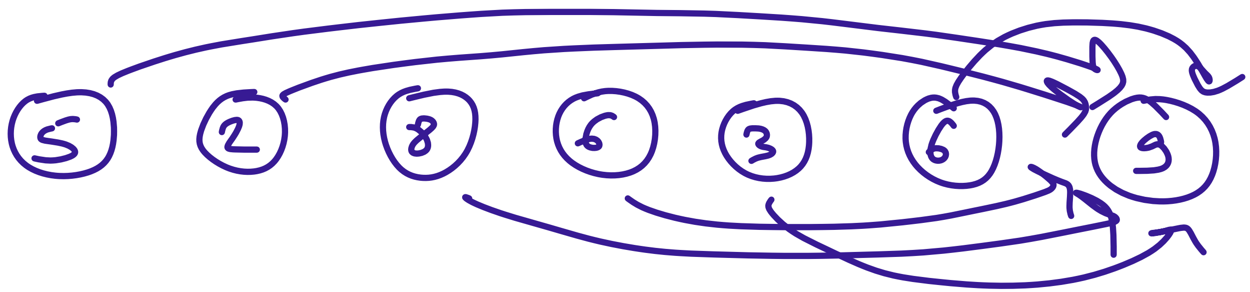
$= \max\{2, 1, 1\}+1 =$

$= 3$

Esempio

$A = \langle 5, 2, 8, 6, 3, 6, 9, 7 \rangle$

	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	



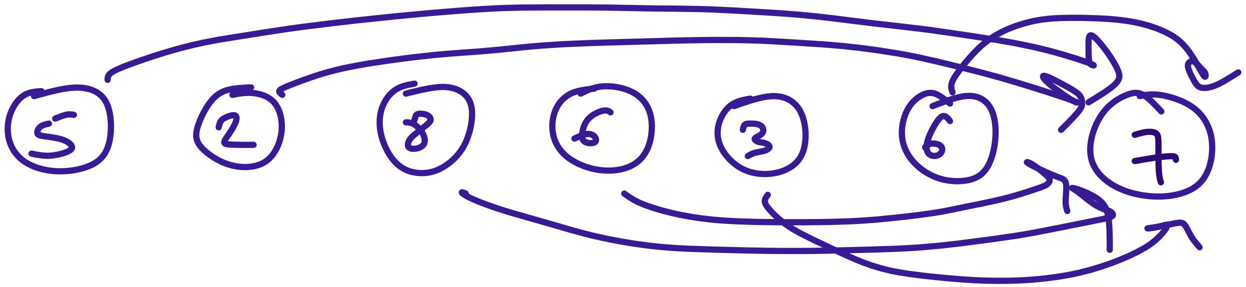
$$L[7] = \max\{L[6], L[5], L[4], L[3], L[2], L[1]\} + 1 =$$

$$= \max\{3, 2, 2, 2, 1, 1\} + 1 = 4$$

Esempio

$A = \langle 5, 2, 8, 6, 3, 6, 9, 7 \rangle$

	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4



$$L[8] = \max\{L[6], L[5], L[4], L[3], L[2], L[1]\} + 1 =$$

$$= \max\{3, 2, 2, 2, 1, 1\} + 1 = 4$$

Esempio

$A = \langle 5, 2, 8, 6, 3, 6, 9, 7 \rangle$

	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4

SOLUZIONI OTTIME

$A_1 = \langle 2, 3, 6, 9 \rangle$

$A_2 = \langle 2, 3, 6, 7 \rangle$

LUNGHEZZA
4

Longest Increasing Subsequence (LSI)

```
LSI(A)
costruire il DAG G a partire da A
for j = 1 to n do
 L[j] := 1
for j = 2 to n do
```
$$L[j] = \max_{(i,j)\in E} \{L[i]\} + 1$$
```
return maxj L[j]
```

In questo modo abbiamo solo calcolato la lunghezza della sottosequenza più lunga, non abbiamo la sottosequenza

Longest Increasing Subsequence (LSI)

```
LSI(A)
costruire il DAG G a partire da A
for j = 1 to n do
 L[j] := 1
for j = 2 to n do
```
$$L[j] = \max_{(i,j)\in E} \{L[i]\} + 1$$
```
return maxj L[j]
```

In questo modo abbiamo solo calcolato la lunghezza della sottosequenza più lunga, non abbiamo la sottosequenza

⇩

teniamo traccia di come arrivare alle soluzioni ottime dei sottoproblemi

Longest Increasing Subsequence (LSI)

```
LSI(A)
costruire il DAG G a partire da A
for j = 1 to n do
 L[j] := 1
for j = 2 to n do
```
$$L[j] = \max_{(i,j)\in E} \{L[i]\} + 1$$
```
return maxj L[j]
```

Teniamo traccia del predecessore di un nodo
nel cammino minimo

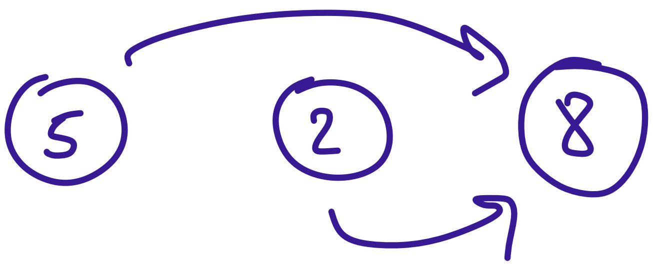
↑

teniamo traccia di come arrivare alle soluzioni ottime
dei sottoproblemi

Esempio

$A = (5, 2, 8, 6, 3, 6, 9, 7)$

	1	2	3	4	5	6	7	8
L	1	1	2					
PREV	/	/	1					



$L[3] = 2$

$PREV[3] = 1$ OPPURE 2

Esempio

$A = \langle 5, 2, 8, 6, 3, 6, 9, 7 \rangle$

	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	
PREV	/	/	1	1	2	5	6	

7

$L[8] = L[6] + 1$

Esempio

$A = (5, 2, 8, 6, 3, 6, 9, 7)$

	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4
PREV	/	/	1	1	2	5	6	6

7

$L[8] = L[6] + 1$

Esempio

$A = \langle 5, 2, 8, 6, 3, 6, 9, 7 \rangle$

	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4
PREV	/	/	1	1	2	5	6	6

Esempio

$A = \langle 5, 2, 8, 6, 3, 6, 9, 7 \rangle$

	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4
PREV	/	/	1	1	2	5	6	6

Esempio

$A = (5, 2, 8, 6, 3, 6, 9, 7)$

	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4
PREV	/	/	1	1	2	5	6	6

Esempio

$A = (5, 2, 8, 6, 3, 6, 9, 7)$

	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4
PREV	/	/	1	1	2	5	6	6

Esempio

$A = \langle 5, 2, 8, 6, 3, 6, 9, 7 \rangle$

	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4
PREV	/	/	1	1	2	5	6	6

$LIS = \langle 2, 3, 6, 7 \rangle$

Longest Increasing Subsequence (LSI)

```
LSI(A[1..n])
costruire il DAG G a partire da A
for j = 1 to n do
   L[j] := 1
   prev[j] := 0
for j = 2 to n do
```

$$L[j] = \max_{(i,j)\in E} \{L[i]\} + 1$$

```
   prev[j] := indice che ha massimizzato L[j]
lis := new_stack()
k := indice del valore massimo in L[]
while k>0 do
   push(lis,A[k])
   k := prev[k]
return lis
```

Programmazione Dinamica

La **PROGRAMMAZIONE DINAMICA**
può essere applicata se il **PROBLEMA** ha una
SOTTOSTUTTURA OTTIMA:

- È possibile combinare le soluzioni ottime dei sottoproblemi per trovare la soluzione di un problema più grande
- Le scelte fatte per risolvere i sottoproblemi in modo ottimo non devono essere modificate quando la soluzione al sottoproblema diventa una parte della soluzione al problema più grande

Affinché l'algoritmo abbia **costo computazionale polinomiale** occorre che:

- il numero di sottoproblemi da risolvere sia polinomiale
- il tempo per combinare soluzioni dei sottoproblemi sia polinomiale

Si memorizzano le soluzioni dei sottoproblemi in una matrice, senza sapere quali saranno necessarie e quali no

Programmazione Dinamica

Per il problema della LIS

- Si memorizzano i **valori** delle soluzioni ottime ai sottoproblemi per utilizzarli quando (e se) serve
- Si memorizzano le **scelte** che hanno portato alle soluzioni ottime dei sottoproblemi, per poter ricostruire la soluzione ottima al problema più grande
- Si deve determinare un **ordine** in cui risolvere i sottoproblemi per dimensione crescente
- Si deve determinare la soluzione ottima dei sottoproblemi più piccoli (**casi base**), per i quali non è possibile una ulteriore suddivisione

Valori in L[1..n]

Valori in prev[1..n]

Ordine in cui compaiono i valori nella sequenza di INPUT (ordine topologico DAG)

Se il nodo del DAG in posizione j non ha archi entranti L[j] = 1 e prev[j] = NIL

UNIMORE