

### Paradigmi algoritmici

**Greedy.** Costruisce la soluzione incrementalmente, ottimizzando qualche criterio locale.

**Divide-and-conquer.** Divide un problema in diversi sottoproblemi, risolve ogni sottoproblema indipendentemente, e combina le soluzioni ai sottoproblemi per costruire la soluzione al problema originale.

**Programmazione dinamica.** Divide il problema in una serie di sottoproblemi che si sovrappongono, e costruisce le soluzioni dai sottoproblemi più piccoli a quelli sempre più grandi.

2

### Programmazione Dinamica: storia

**Richard Bellman.** E' stato il pioniere dello studio sistematico della programmazione dinamica negli anni '50.

#### Etimologia.

- Programmazione dinamica = pianificazione nel tempo.  
programmazione matematica e non programmazione informatica
- Assistant Secretary of Air Force era ostile alla ricerca matematica.
- Bellman pensò ad un nome altisonante per evitarne il confronto.
  - "it's impossible to use dynamic in a pejorative sense"
  - "something not even a Congressman could object to"



3

### Programmazione Dinamica: Applicazioni

#### Aree.

- Bioinformatica.
- Teoria dei Controlli.
- Teoria dell' Informazione.
- Ricerca Operativa.
- Informatica: teoria, grafica, Intelligenza Artificiale, ....

#### Alcuni algoritmi di programmazione dinamica famosi.

- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- Smith-Waterman for sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

4

### Programmazione Dinamica

Algoritmi che vedremo:

- Weighted Interval Scheduling
- Segmented Least Squares
- Knapsack Problem
- RNA Secondary Structure
- Sequence Alignment
- Shortest Paths in a Graph

Esercizio:

- Longest Common Subsequence

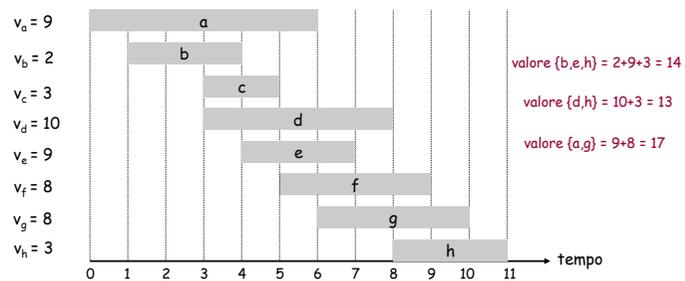
5

## 6.1 Weighted Interval Scheduling

### Schedulazione degli intervalli pesati

Problema della schedulazione degli intervalli pesati.

- Job  $j$  inizia ad  $s_j$ , finisce a  $f_j$ , ed ha peso / valore  $v_j$ .
- Due job **compatibili** se non hanno intersezione.
- Obiettivo: trovare sottoinsieme con massimo peso di job mutuamente compatibili.



7

### Schedulazione intervalli senza pesi

Ricordiamo. L'algoritmo greedy risolve il problema se tutti i pesi sono 1.

- Considerare job in ordine crescente di tempo di fine.
- Aggiungere job al sottoinsieme della soluzione se è compatibile con i job scelti precedentemente.

Osservazione. L'algoritmo greedy può dare soluzioni non ottime se ci sono pesi arbitrari.

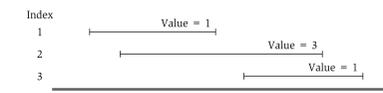
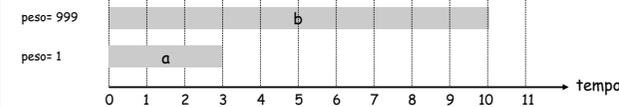


Figure 6.1 A simple instance of weighted interval scheduling.



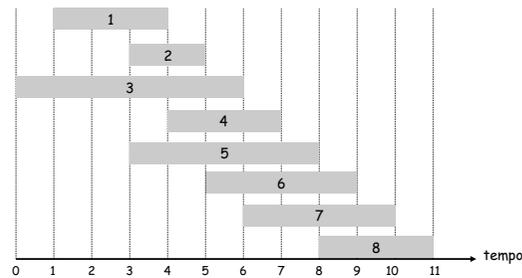
8

Schedulazione degli intervalli pesati

Notazione. Ordinare job per tempo di fine:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

Definizione.  $p(j)$  = il più grande  $i < j$  tale che job  $i$  è compatibile con  $j$ .

Esempio:  $p(8) = 5, p(7) = 3, p(2) = 0$ .



9

Schedulazione degli intervalli pesati

Notazione. Ordinare job per tempo di fine:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

Definizione.  $p(j)$  = il più grande  $i < j$  tale che job  $i$  è compatibile con  $j$ .

Esempio:

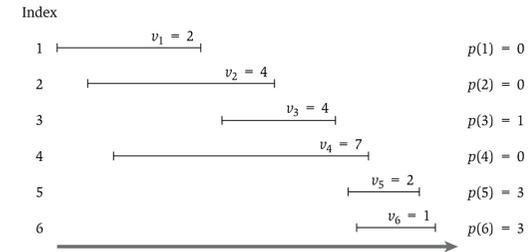


Figure 6.2 An instance of weighted interval scheduling with the functions  $p(j)$  defined for each interval  $j$ .

10

Programmazione dinamica: Scelta binaria

Notazione.  $OPT(j)$  = valore della soluzione ottimale al problema con i job  $1, 2, \dots, j$ .

- Caso 1:  $OPT$  contiene job  $j$ .
  - non può contenere job incompatibili  $\{ p(j) + 1, p(j) + 2, \dots, j - 1 \}$
  - deve includere soluzione ottimale al problema che consiste dei rimanenti job compatibili  $1, 2, \dots, p(j)$
- Caso 2:  $OPT$  non contiene job  $j$ .
  - Deve includere soluzione ottimale al problema che consiste dei rimanenti job compatibili  $1, 2, \dots, j-1$

11

Programmazione dinamica: Scelta binaria

Notazione.  $OPT(j)$  = valore della soluzione ottimale al problema con i job  $1, 2, \dots, j$ .

- Caso 1:  $OPT$  contiene job  $j$ .
  - non può contenere job incompatibili  $\{ p(j) + 1, p(j) + 2, \dots, j - 1 \}$
  - deve includere soluzione ottimale al problema che consiste dei rimanenti job compatibili  $1, 2, \dots, p(j)$
- Caso 2:  $OPT$  non contiene job  $j$ .
  - Deve includere soluzione ottimale al problema che consiste dei rimanenti job compatibili  $1, 2, \dots, j-1$

$$OPT(j) = \begin{cases} 0 & \text{se } j=0 \\ \max \{ v_j + OPT(p(j)), OPT(j-1) \} & \text{altrimenti} \end{cases}$$

12

Schedulazione degli intervalli pesati: Algoritmo di forza bruta

Algoritmo di forza bruta.

```

Input:  $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$ 
Ordinare job per tempo di fine  $f_1 \leq f_2 \leq \dots \leq f_n$ .
Computare  $p(1), p(2), \dots, p(n)$ 
Compute-Opt(j) {
  if (j = 0)
    return 0
  else
    return  $\max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1))$ 
}
    
```

13

Schedulazione degli intervalli pesati: Algoritmo di forza bruta

Osservazione. Algoritmo ricorsivo non è efficiente perchè ci sono molti sottoproblemi ridondanti

Esempio.

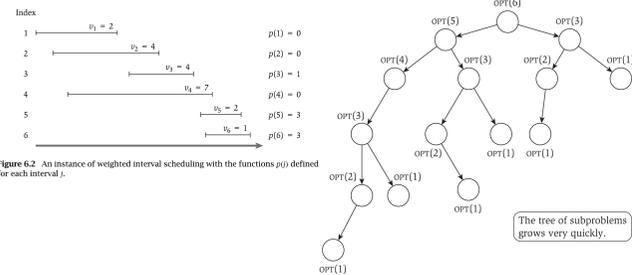


Figure 6.2 An instance of weighted interval scheduling with the functions  $p(j)$  defined for each interval  $j$ .

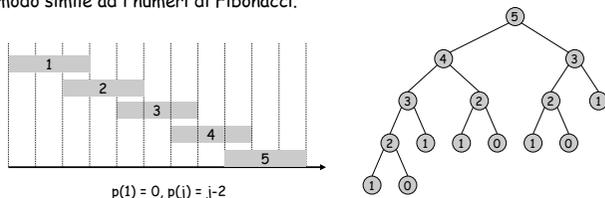
Figure 6.3 The tree of subproblems called by Compute-Opt on the problem instance of Figure 6.2.

14

Schedulazione degli intervalli pesati: Algoritmo di forza bruta

Osservazione. Algoritmo ricorsivo non è efficiente perchè ci sono molti sottoproblemi ridondanti  $\Rightarrow$  complessità esponenziale.

Esempio. Numero di chiamate ricorsive per una famiglia di istanze cresce in modo simile ad i numeri di Fibonacci.



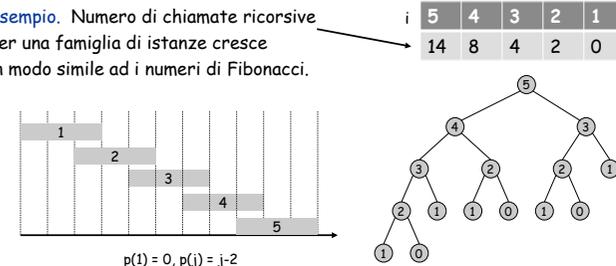
$T(n) = T(n-1) + T(n-2) + O(1)$

15

Schedulazione degli intervalli pesati: Algoritmo di forza bruta

Osservazione. Algoritmo ricorsivo non è efficiente perchè ci sono molti sottoproblemi ridondanti  $\Rightarrow$  complessità esponenziale.

Esempio. Numero di chiamate ricorsive per una famiglia di istanze cresce in modo simile ad i numeri di Fibonacci.



$T(n) = T(n-1) + T(n-2) + O(1)$

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

$F_i = F_{i-1} + F_{i-2}$   
 $F_1 = 1 \quad F_0 = 0$

$F_i = \frac{\varphi^i - \hat{\varphi}^i}{\sqrt{5}}$

$\varphi = \frac{1 + \sqrt{5}}{2} = 1,61803\dots$   
 $\hat{\varphi} = \frac{1 - \sqrt{5}}{2} = -0,61803\dots$

16

Schedulazione degli intervalli pesati: Annotazione

**Annotazione (Memoization).** Memorizzare i risultati per ogni sottoproblema e verificare quando necessario se un sottoproblema è già stato risolto.

**Input:**  $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$

**Ordinare** job per tempo di fine  $f_1 \leq f_2 \leq \dots \leq f_n$ .  
**Compute**  $p(1), p(2), \dots, p(n)$

**for**  $j = 1$  to  $n$   
 $M[j] = \text{empty}$  ← array globale

```

M-Compute-Opt(j)
  If  $j = 0$  then
    Return 0
  Else if  $M[j]$  is not empty then
    Return  $M[j]$ 
  Else
    Define  $M[j] = \max(v_j + M\text{-Compute-Opt}(p(j)), M\text{-Compute-Opt}(j - 1))$ 
    Return  $M[j]$ 
  Endif
    
```

17

Schedulazione degli intervalli pesati: Complessità

**Claim.** L' algoritmo con l'annotazione ha complessità  $O(n \log n)$ .

- Ordinamento per tempo di fine:  $O(n \log n)$ .
- Computazione di  $p(\cdot)$ :  $O(n)$  dopo un ordinamento per tempo di inizio.

← esercizio da fare a casa

18

Schedulazione degli intervalli pesati: Complessità

**Claim.** L' algoritmo con l' annotazione ha complessità  $O(n \log n)$ .

- Ordinamento per tempo di fine:  $O(n \log n)$ .
- Computazione di  $p(\cdot)$ :  $O(n)$  dopo un ordinamento per tempo di inizio.

← esercizio da fare a casa

- $M\text{-Compute-Opt}(j)$ : ogni chiamata richiede tempo  $O(1)$  e fa una delle due
  - ritorna un valore già calcolato  $M[j]$
  - calcola una nuovo valore  $M[j]$  e fa due chiamate ricorsive

- Misura del progresso  $\Phi =$  numero di valori nonvuoti di  $M[\cdot]$ .
  - inizialmente  $\Phi = 0$ , poi  $\Phi \leq n$ .
  - incremento  $\Phi$  di 1  $\Rightarrow$  al massimo totale di  $O(n)$  chiamate ricorsive.

- Tempo totale di  $M\text{-Compute-Opt}(n)$  è  $O(n)$ .

**Osservazione.** Tempo  $O(n)$  se job sono ordinati per tempo di inizio e per tempo di fine.

19

Annotazione automatica

**Annotazione automatica.** Molti linguaggi di programmazione funzionali (e.g., Lisp) hanno un supporto built-in support per l' annotazione.

Non succede nei linguaggi imperativi (e.g., Java)?

```

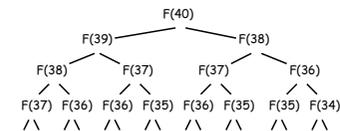
(defun F (n)
  (if (<= n 1)
      n
      (+ (F (- n 1)) (F (- n 2)))))
    
```

Lisp (efficiente)

```

static int F(int n) {
  if (n <= 1) return n;
  else return F(n-1) + F(n-2);
}
    
```

Java (esponenziale)



20

Schedulazione degli intervalli pesati: Trovare una soluzione

**Domanda.** Algoritmo di programmazione dinamica computa il valore ottimale. E se voglio una soluzione?

**Risposta.** Necessarie operazioni di post-processing.

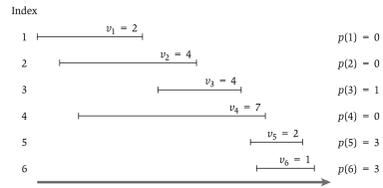


Figure 6.2 An instance of weighted interval scheduling with the functions  $p(j)$  defined for each interval  $j$ .

$$M = \begin{matrix} & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \begin{matrix} \boxed{0} & \boxed{2} & \boxed{4} & \boxed{6} & \boxed{7} & \boxed{8} & \boxed{8} \end{matrix} \end{matrix}$$

21

Schedulazione degli intervalli pesati: Trovare una soluzione

**Domanda.** Algoritmo di programmazione dinamica computa il valore ottimale. E se voglio una soluzione?

**Risposta.** Necessarie operazioni di post-processing.

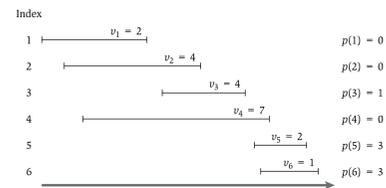


Figure 6.2 An instance of weighted interval scheduling with the functions  $p(j)$  defined for each interval  $j$ .

E' 6 nella soluzione ottimale?

$$M = \begin{matrix} & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \begin{matrix} \boxed{0} & \boxed{2} & \boxed{4} & \boxed{6} & \boxed{7} & \boxed{8} & \boxed{8} \end{matrix} \end{matrix}$$

22

Schedulazione degli intervalli pesati: Trovare una soluzione

**Domanda.** Algoritmo di programmazione dinamica computa il valore ottimale. E se voglio una soluzione?

**Risposta.** Necessarie operazioni di post-processing.

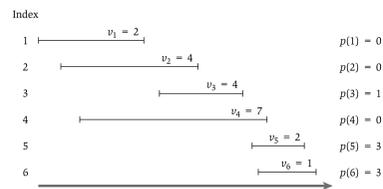


Figure 6.2 An instance of weighted interval scheduling with the functions  $p(j)$  defined for each interval  $j$ .

E' 6 nella soluzione ottimale? NO

$$M = \begin{matrix} & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \begin{matrix} \boxed{0} & \boxed{2} & \boxed{4} & \boxed{6} & \boxed{7} & \boxed{8} & \boxed{8} \end{matrix} \end{matrix}$$

$$OPT(j) = \max \{ v_j + OPT(p(j)), OPT(j-1) \}$$

$$OPT(6) = \max \{ v_6 + OPT(p(6)), OPT(5) \} = \max \{ 1 + OPT(3), OPT(5) \} = \max \{ 1 + 6, 8 \}$$

23

Schedulazione degli intervalli pesati: Trovare una soluzione

**Domanda.** Algoritmo di programmazione dinamica computa il valore ottimale. E se voglio una soluzione?

**Risposta.** Necessarie operazioni di post-processing.

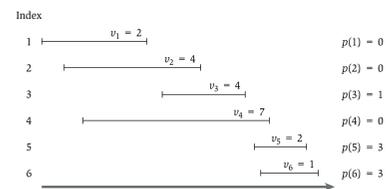


Figure 6.2 An instance of weighted interval scheduling with the functions  $p(j)$  defined for each interval  $j$ .

E' 5 nella soluzione ottimale?

$$M = \begin{matrix} & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \begin{matrix} \boxed{0} & \boxed{2} & \boxed{4} & \boxed{6} & \boxed{7} & \boxed{8} & \boxed{8} \end{matrix} \end{matrix}$$

24

Schedulazione degli intervalli pesati: Trovare una soluzione

**Domanda.** Algoritmo di programmazione dinamica computa il valore ottimale. E se voglio una soluzione?

**Risposta.** Necessarie operazioni di post-processing.

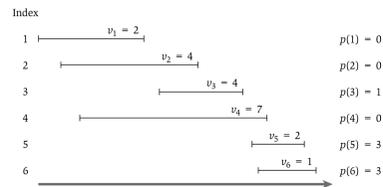


Figure 6.2 An instance of weighted interval scheduling with the functions  $p(j)$  defined for each interval  $j$ .

E' 5 nella soluzione ottimale? SI

0	1	2	3	4	5	6
0	2	4	6	7	8	8

$$OPT(j) = \max \{ v_j + OPT(p(j)), OPT(j-1) \}$$

$$OPT(5) = \max \{ v_5 + OPT(p(5)), OPT(4) \} = \max \{ 2 + OPT(3), OPT(4) \} = \max \{ 2 + 6, 7 \}$$

25

Schedulazione degli intervalli pesati: Trovare una soluzione

**Domanda.** Algoritmo di programmazione dinamica computa il valore ottimale. E se voglio una soluzione?

**Risposta.** Necessarie operazioni di post-processing.

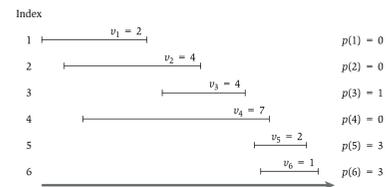


Figure 6.2 An instance of weighted interval scheduling with the functions  $p(j)$  defined for each interval  $j$ .

E' 3 nella soluzione ottimale?

0	1	2	3	4	5	6
0	2	4	6	7	8	8

26

Schedulazione degli intervalli pesati: Trovare una soluzione

**Domanda.** Algoritmo di programmazione dinamica computa il valore ottimale. E se voglio una soluzione?

**Risposta.** Necessarie operazioni di post-processing.

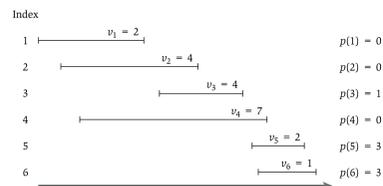


Figure 6.2 An instance of weighted interval scheduling with the functions  $p(j)$  defined for each interval  $j$ .

E' 3 nella soluzione ottimale? SI

0	1	2	3	4	5	6
0	2	4	6	7	8	8

$$OPT(j) = \max \{ v_j + OPT(p(j)), OPT(j-1) \}$$

$$OPT(3) = \max \{ v_3 + OPT(p(3)), OPT(2) \} = \max \{ 4 + OPT(1), OPT(2) \} = \max \{ 4 + 2, 4 \}$$

27

Schedulazione degli intervalli pesati: Trovare una soluzione

**Domanda.** Algoritmo di programmazione dinamica computa il valore ottimale. E se voglio una soluzione?

**Risposta.** Necessarie operazioni di post-processing.

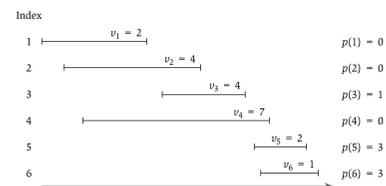


Figure 6.2 An instance of weighted interval scheduling with the functions  $p(j)$  defined for each interval  $j$ .

E' 1 nella soluzione ottimale? SI

0	1	2	3	4	5	6
0	2	4	6	7	8	8

$$OPT(j) = \max \{ v_j + OPT(p(j)), OPT(j-1) \}$$

$$OPT(1) = \max \{ v_1 + OPT(p(1)), OPT(0) \} = \max \{ 2 + OPT(0), OPT(0) \} = \max \{ 2 + 0, 0 \}$$

28

Schedulazione degli intervalli pesati: Trovare una soluzione

Domanda. Algoritmo di programmazione dinamica computa il valore ottimale. E se voglio una soluzione?

Risposta. Necessarie operazioni di post-processing.

```

Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
  if (j = 0)
    output nothing
  else if (vj + M[p(j)] > M[j-1])
    print j
    Find-Solution(p(j))
  else
    Find-Solution(j-1)
}
    
```

Numero di chiamate ricorsive  $\leq n \Rightarrow O(n)$ .

29

Schedulazione degli intervalli pesati: Approccio bottom-Up

Programmazione dinamica bottom-up.

```

Input: n, s1, ..., sn, f1, ..., fn, v1, ..., vn

Ordinare job per tempo di fine f1 ≤ f2 ≤ ... ≤ fn.

Computa p(1), p(2), ..., p(n)

Iterative-Compute-Opt {
  M[0] = 0
  for j = 1 to n
    M[j] = max(vj + M[p(j)], M[j-1])
}
    
```

Complessità  $O(n)$ .

30

Schedulazione degli intervalli pesati: Approccio bottom-Up

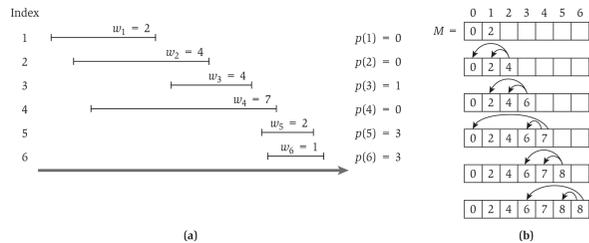


Figure 6.5 Part (b) shows the iterations of Iterative-Compute-Opt on the sample instance of Weighted Interval Scheduling depicted in part (a).

31

Esercizio: calcolo p(j)

Computazione di  $p(\cdot)$ :  $O(n)$  dopo un ordinamento per tempo di inizio.

Quindi, dati i job ordinati per tempo di inizio, come computare in tempo lineare i valori  $p(j)$ ?

Primo algoritmo:

```

Input: n, s1, ..., sn, f1, ..., fn

job ordinati per tempo di fine f1 ≤ f2 ≤ ... ≤ fn.

Computa-valori-p {
  f0=0
  for i=1 to n
    Calcola il max j tale che fj ≤ si
    p(i)=j
}
    
```

32

Esercizio: calcolo p(j)

Computazione di p(-): O(n) dopo un ordinamento per tempo di inizio.

Quindi, dati i job ordinati per tempo di inizio, come computare in tempo lineare i valori p(j)?

Primo algoritmo:

```

Input: n, s1, ..., sn, f1, ..., fn

job ordinati per tempo di fine f1 ≤ f2 ≤ ... ≤ fn.

Computa-valori-p {
  f0=0
  for i=1 to n
    Calcola il max j tale che fj ≤ si
    p(i)=j
}
    
```

Complessità?

33

Esercizio: calcolo p(j)

Computazione di p(-): O(n) dopo un ordinamento per tempo di inizio.

Quindi, dati i job ordinati per tempo di inizio, come computare in tempo lineare i valori p(j)?

```

Input: n, s1, ..., sn, f1, ..., fn, n

job ordinati per tempo di fine f1 ≤ f2 ≤ ... ≤ fn.
job ordinati per tempo di inizio sn(1) ≤ sn(2) ≤ ... ≤ sn(n).

Computa-valori-p {
  j=0
  f0=0
  for i=1 to n
    Calcola il max j tale che fj ≤ sn(i)
    p(n(i))=j
}
    
```

34

Esercizio: calcolo p(j)

Computazione di p(-): O(n) dopo un ordinamento per tempo di inizio.

Quindi, dati i job ordinati per tempo di inizio, come computare in tempo lineare i valori p(j)?

```

Input: n, s1, ..., sn, f1, ..., fn, n

job ordinati per tempo di fine f1 ≤ f2 ≤ ... ≤ fn.
job ordinati per tempo di inizio sn(1) ≤ sn(2) ≤ ... ≤ sn(n).

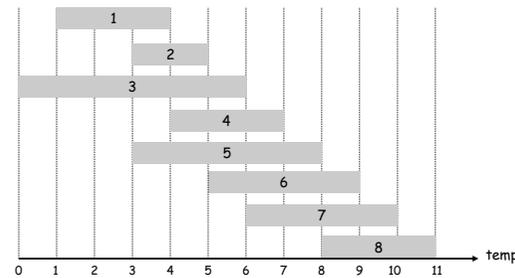
Computa-valori-p {
  j=0
  f0=0
  for i=1 to n
    Calcola il max j tale che fj ≤ sn(i)
    p(n(i))=j
}
    
```

O(n)

Non può essere più piccolo del valore j precedente!

35

Esercizio: calcolo p(j), esempio



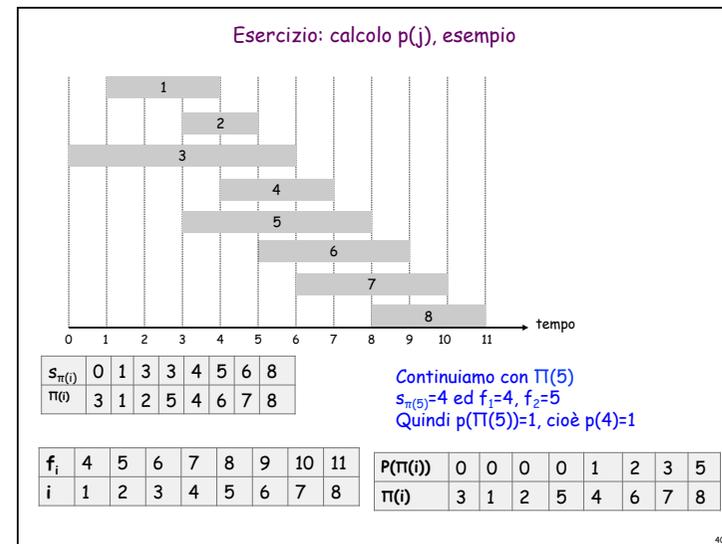
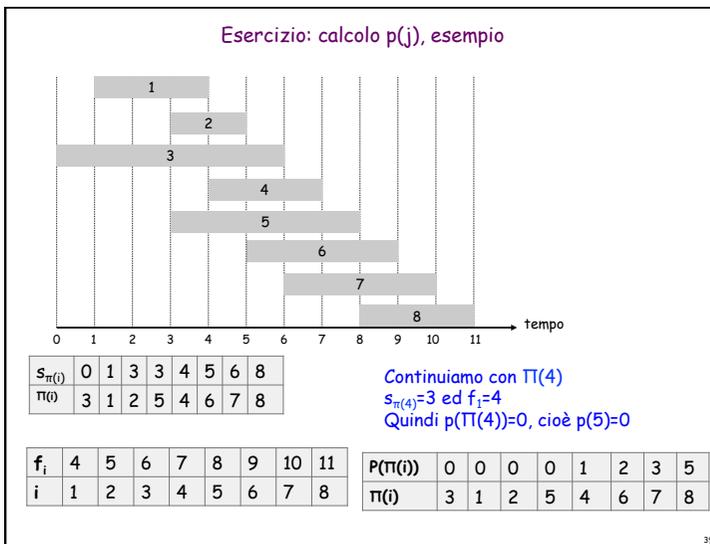
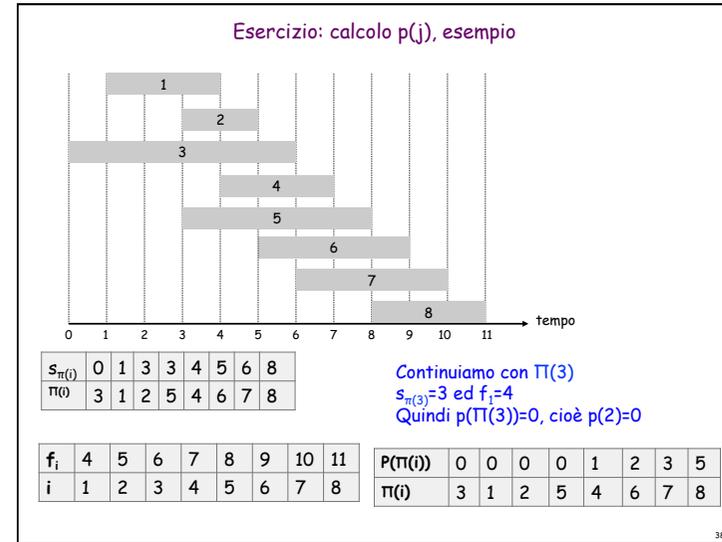
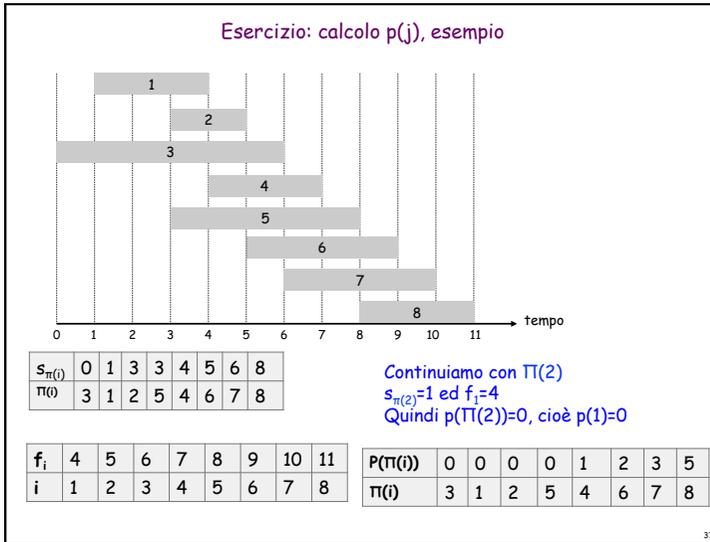
s <sub>n(i)</sub>	0	1	3	3	4	5	6	8
π(i)	3	1	2	5	4	6	7	8

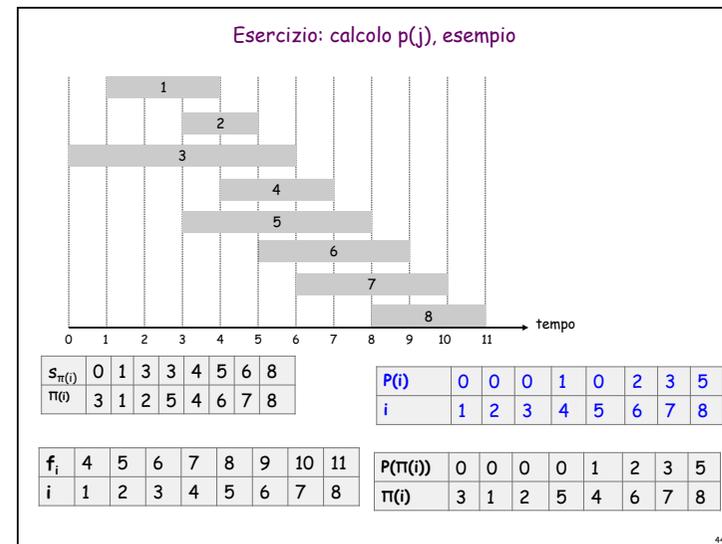
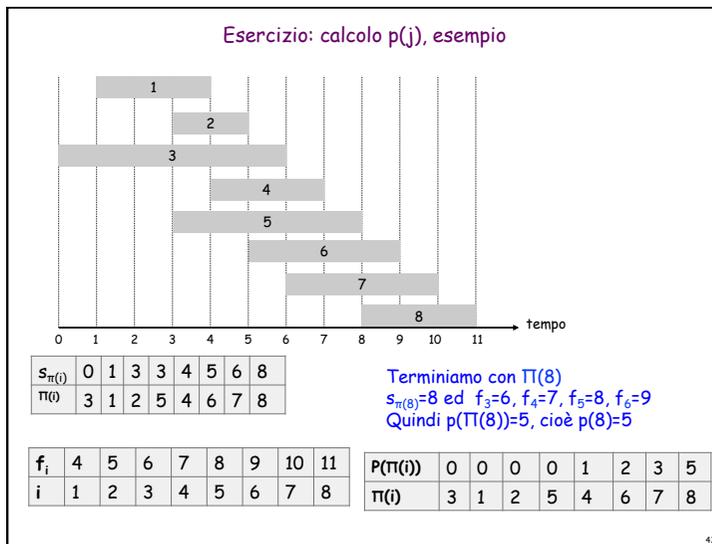
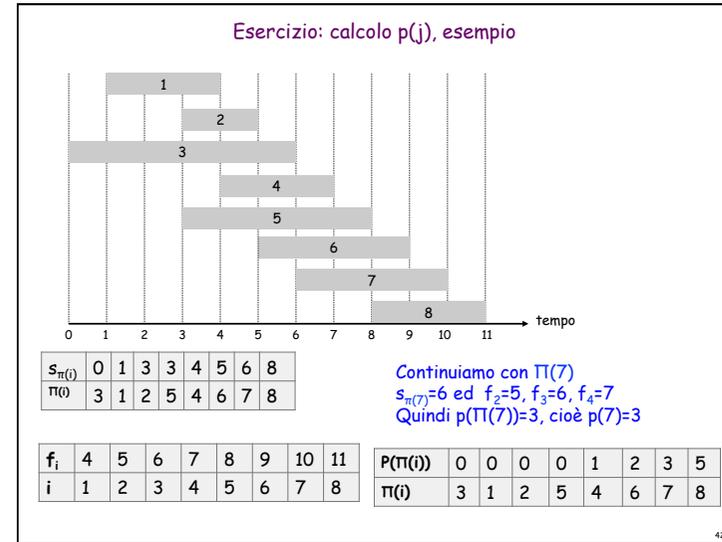
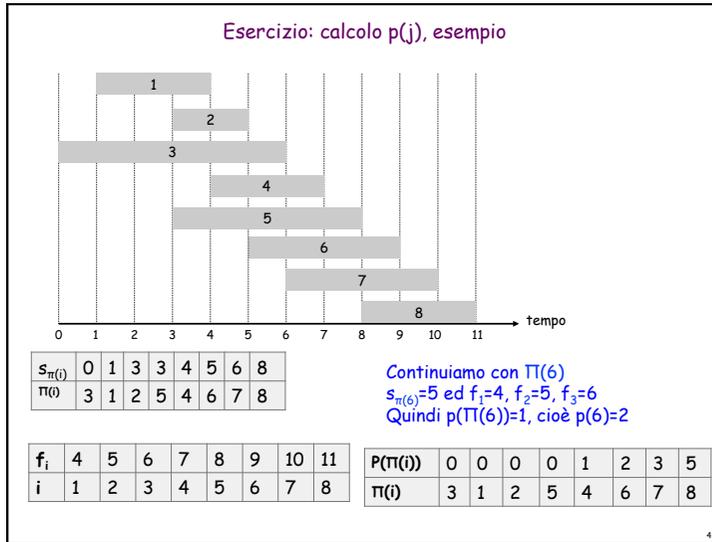
Iniziamo con π(1)  
 s<sub>n(1)</sub>=0 ed f<sub>1</sub>=4  
 Quindi: p(π(1))=0, cioè p(3)=0

f <sub>i</sub>	4	5	6	7	8	9	10	11
i	1	2	3	4	5	6	7	8

P(π(i))	0	0	0	0	1	2	3	5
π(i)	3	1	2	5	4	6	7	8

36



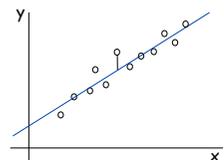


## 6.3 Segmented Least Squares

### Segmented Least Squares

**Least squares.**

- Problema fondamentale in statistica ed analisi numerica.
- Dati n punti nel piano:  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ .
- Trovare linea  $y = ax + b$  che minimizza la somma dell'errore quadratico:



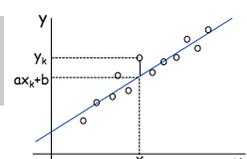
46

### Segmented Least Squares

**Least squares.**

- Problema fondamentale in statistica ed analisi numerica.
- Dati n punti nel piano:  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ .
- Trovare linea  $y = ax + b$  che minimizza la somma dell'errore quadratico:

$$\text{Errore}("y = ax + b", \text{punti}) = \sum_{k=1}^n (y_k - ax_k - b)^2$$



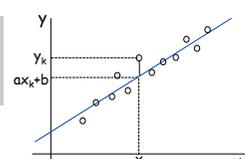
47

### Segmented Least Squares

**Least squares.**

- Problema fondamentale in statistica ed analisi numerica.
- Dati n punti nel piano:  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ .
- Trovare linea  $y = ax + b$  che minimizza la somma dell'errore quadratico:

$$\text{Errore}("y = ax + b", \text{punti}) = \sum_{k=1}^n (y_k - ax_k - b)^2$$



**Soluzione.** Sappiamo che l'errore minimo si ha quando

$$a = \frac{n \sum_k x_k y_k - (\sum_k x_k)(\sum_k y_k)}{n \sum_k x_k^2 - (\sum_k x_k)^2}, \quad b = \frac{\sum_k y_k - a \sum_k x_k}{n}$$

48

Insieme di punti su due linee

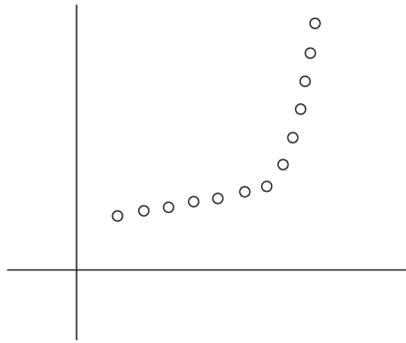


Figure 6.7 A set of points that lie approximately on two lines.

49

Insieme di punti su tre linee

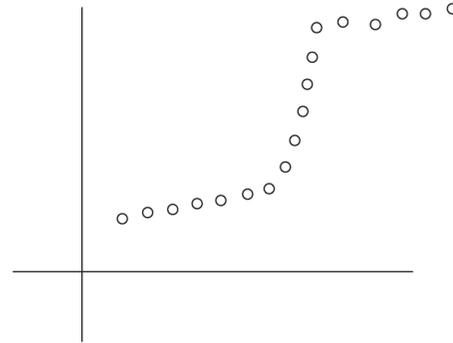


Figure 6.8 A set of points that lie approximately on three lines.

50

Segmented Least Squares

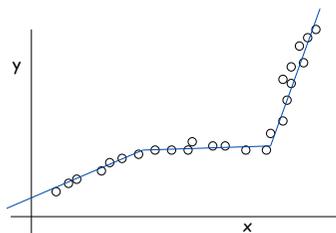
Segmented least squares.

- Punti si trovano approssimativamente su una sequenza di segmenti.
- Dati  $n$  punti nel piano  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$  con  $x_1 < x_2 < \dots < x_n$ , trovare una sequenza di linee che minimizza  $f(x)$ .

Domanda. Qual'è una scelta ragionevole per  $f(x)$  per bilanciare accuratezza e parsominia?

↑  
numero di linee

↑  
bontà dell'approssimazione

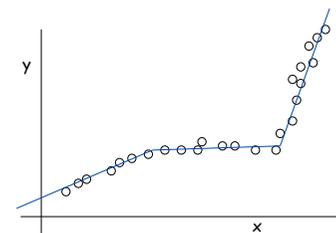


51

Segmented Least Squares

Segmented least squares.

- Punti si trovano approssimativamente su una sequenza di segmenti.
- Dati  $n$  punti nel piano  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$  con  $x_1 < x_2 < \dots < x_n$ , trovare una sequenza di linee che minimizza :
  - la somma delle somme degli errori quadratici  $E$  in ogni segmento
  - il numero delle linee  $L$
- Funzione tradeoff:  $E + cL$ , per qualche costante  $c > 0$ .



52

Programmazione dinamica: Multiway Choice

Notazione.

- OPT(j) = minimo costo per punti  $p_1, p_2, \dots, p_j$ . [con OPT(0)=0]
- $e(i, j)$  = minima somma di errori quadratici per punti  $p_i, p_{i+1}, \dots, p_j$ .

Per computare OPT(j):

- L'ultimo segmento usa punti  $p_i, p_{i+1}, \dots, p_j$  per qualche i.
- Costo =  $e(i, j) + c + OPT(i-1)$ .

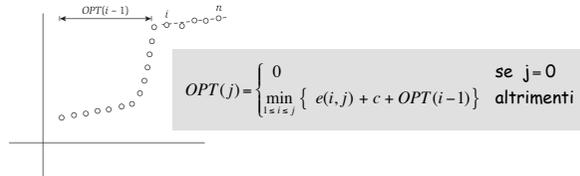


Figure 6.9 A possible solution: a single line segment fits points  $p_i, p_{i+1}, \dots, p_j$ , and then an optimal solution is found for the remaining points  $p_1, p_2, \dots, p_{i-1}$ .

53

Segmented Least Squares: Algoritmo

```

INPUT: n, P1, ..., Pn, c
Segmented-Least-Squares() {
  M[0] = 0
  for j = 1 to n
    for i = 1 to j
      compute il minimo errore quadratico eij
      per il segmento P1, ..., Pj
  for j = 1 to n
    M[j] = min_{1 <= i <= j} (eij + c + M[i-1])
  return M[n]
}
    
```

Complessità.  $O(n^3)$ .

- Collo di bottiglia = computazione di  $e(i, j)$  per  $O(n^2)$  coppie,  $O(n)$  per coppia usando la formula precedente

54

Segmented Least Squares: Algoritmo

```

INPUT: n, P1, ..., Pn, c
Segmented-Least-Squares() {
  M[0] = 0
  for j = 1 to n
    for i = 1 to j
      compute the least square error eij for
      the segment P1, ..., Pj
  for j = 1 to n
    M[j] = min_{1 <= i <= j} (eij + c + M[i-1])
  return M[n]
}
    
```

può essere migliorato:  $O(n^2)$  pre-computando varie statistiche  
 Esercizio da fare a casa  
 Idea: calcolare valori  $(i, j)$  per  $j=i-1, j=i-2, \dots$   
 valori  $(i, j)$  da valori  $(i, j-1)$  in  $O(1)$

Complessità:  $O(n^3)$

Collo di bottiglia = computazione di  $e(i, j)$  per  $O(n^2)$  coppie,  $O(n)$  per coppia usando la formula precedente

55

Esercizio: calcolare tutti i valori  $e(i, j)$  in  $O(n^2)$

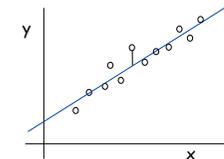
Algoritmo di forza bruta:  $O(n^3)$

Suggerimento per algoritmo  $O(n^2)$ :

- calcolare valori  $e(i, j)$  per  $j=i-1, j=i-2, \dots$
- calcolare  $e(i, j)$  da  $e(i, j-1)$  in  $O(1)$

$$e(i, j) = \sum_{k=i}^j (y_k - ax_k - b)^2$$

$$a = \frac{n \sum x_k y_k - (\sum x_k)(\sum y_k)}{n \sum x_k^2 - (\sum x_k)^2}, \quad b = \frac{\sum y_k - a \sum x_k}{n}$$



56

Esercizio: calcolare tutti i valori  $e(i,j)$  in  $O(n^2)$

Algoritmo di forza bruta:  $O(n^3)$

Suggerimento per algoritmo  $O(n^2)$ :

□ calcolare valori  $(i,j)$  per  $j-i=1, j-i=2, \dots$

□ calcolare valori  $(i,j)$  da quelli  $(i,j-1)$  in  $O(1)$

$$e(i,j) = \sum_{k=i}^j (y_k - ax_k - b)^2$$

$$a = \frac{n \sum_k x_k y_k - (\sum_k x_k)(\sum_k y_k)}{n \sum_k x_k^2 - (\sum_k x_k)^2}, \quad b = \frac{\sum_k y_k - a \sum_k x_k}{n}$$

Soluzione: calcolare tutte le somme  $\sum_{k=i}^j x_k, \sum_{k=i}^j y_k, \sum_{k=i}^j x_k y_k, \sum_{k=i}^j x_k^2, \sum_{k=i}^j y_k^2$  in  $O(n^2)$

Scrivere l'errore come

$$\sum_{k=i}^j (y_k - ax_k - b)^2 = \sum_{k=i}^j y_k^2 + a^2 \sum_{k=i}^j x_k^2 + b^2 n - 2a \sum_{k=i}^j x_k y_k - 2b \sum_{k=i}^j y_k + 2ab \sum_{k=i}^j x_k$$

57

Segmented Least Squares: Trovare una soluzione

```

Find-Segments(j)
  If j = 0 then
    Output nothing
  Else
    Find an i that minimizes  $e_{i,j} + C + M[i-1]$ 
    Output the segment  $\{p_i, \dots, p_j\}$  and the result of
      Find-Segments(i-1)
  Endif
    
```

58

## 6.4 Knapsack Problem

### Problema dello zaino

Problema dello zaino.

- Abbiamo n oggetti ed uno "zaino"
- Oggetto i pesa  $w_i > 0$  chilogrammi ed ha valore  $v_i > 0$ .
- Zaino ha una capacità di W chilogrammi.
- Obiettivo: riempire zaino per massimizzare valore totale.

Esempio: { 3, 4 } ha valore 40.

W = 11

oggetti	valore	peso
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Algoritmo greedy: aggiungere oggetto con peso massimo  $w_i$ .

Esempio: { 5, 2, 1 } ha valore = 35  $\Rightarrow$  algoritmo greedy non ottimale.

60

### Problema dello zaino

#### Problema dello zaino.

- Abbiamo  $n$  oggetti ed uno "zaino."
- Oggetto  $i$  pesa  $w_i > 0$  chilogrammi ed ha valore  $v_i > 0$ .
- Zaino ha una capacità di  $W$  chilogrammi.
- Obiettivo: riempire zaino per massimizzare valore totale.

Esempio: { 3, 4 } ha valore 40.

$W = 11$

oggetti	valore	peso
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Algoritmo greedy: aggiungere oggetto con valore massimo  $v_i$ .

Esempio: { 5, 2, 1 } ha valore = 35  $\Rightarrow$  algoritmo greedy non ottimale.

61

### Problema dello zaino

#### Problema dello zaino.

- Abbiamo  $n$  oggetti ed uno "zaino."
- Oggetto  $i$  pesa  $w_i > 0$  chilogrammi ed ha valore  $v_i > 0$ .
- Zaino ha una capacità di  $W$  chilogrammi.
- Obiettivo: riempire zaino per massimizzare valore totale.

Esempio: { 3, 4 } ha valore 40.

$W = 11$

oggetti	valore	peso	val/peso
1	1	1	1
2	6	2	3
3	18	5	3,6
4	22	6	3,66
5	28	7	4

Algoritmo greedy: aggiungere oggetto con rapporto massimo  $v_i / w_i$ .

Esempio: { 5, 2, 1 } ha valore = 35  $\Rightarrow$  algoritmo greedy non ottimale.

62

### Programmazione dinamica: Primo tentativo

Definizione.  $OPT(i) = \max$  valore totale con sottoinsieme oggetti 1, ...,  $i$ .

- Caso 1: OPT non contiene oggetto  $i$ .
  - OPT contiene max valore totale di { 1, 2, ...,  $i-1$  }
- Caso 2: OPT contiene oggetto  $i$ .
  - che l'oggetto  $i$  ci sia nella soluzione non implica immediatamente che altri oggetti non ci siano
  - senza sapere quali altri oggetti ci siano nella soluzione non sappiamo se c'è spazio sufficiente per altri oggetti

Conclusione. Abbiamo bisogno di più sottoproblemi!

63

### Programmazione dinamica: aggiunta nuova variabile

Definizione.  $OPT(i, w) = \max$  valore totale con sottoinsieme oggetti 1, ...,  $i$  con limite di peso  $w$ .

- Case 1: OPT non contiene oggetto  $i$ .
  - OPT contiene max valore totale di { 1, 2, ...,  $i-1$  } con limite di peso  $w$
- Case 2: OPT contiene oggetto  $i$ .
  - Nuovo limite di peso =  $w - w_i$
  - OPT contiene max valore totale di { 1, 2, ...,  $i-1$  } con nuovo limite peso

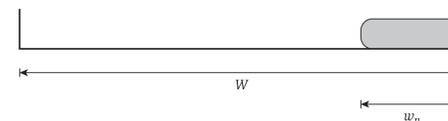


Figure 6.10 After item  $n$  is included in the solution, a weight of  $w_n$  is used up and there is  $W - w_n$  available weight left.

64

Programmazione dinamica: aggiunta nuova variabile

**Definizione.**  $OPT(i, w) = \max$  valore totale con sottoinsieme oggetti 1, ..., i con limite di peso w.

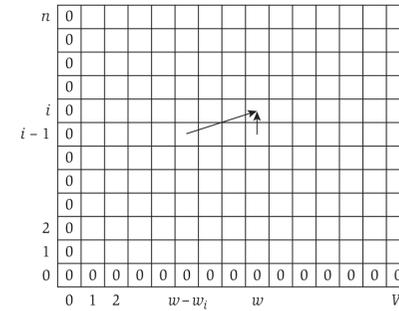
- Case 1: OPT non contiene oggetto i.
  - OPT contiene max valore totale di { 1, 2, ..., i-1 } con limite di peso w
- Case 2: OPT contiene oggetto i.
  - Nuovo limite di peso =  $w - w_i$
  - OPT contiene max valore totale di { 1, 2, ..., i-1 } con nuovo limite di peso

$$OPT(i, w) = \begin{cases} 0 & \text{se } i=0 \\ OPT(i-1, w) & \text{se } w_i > w \\ \max\{OPT(i-1, w), v_i + OPT(i-1, w-w_i)\} & \text{altrimenti} \end{cases}$$

65

Problema dello zaino: approccio bottom-up

Utilizzare un array n-per-W.



**Figure 6.11** The two-dimensional table of OPT values. The leftmost column and bottom row is always 0. The entry for  $OPT(i, w)$  is computed from the two other entries  $OPT(i-1, w)$  and  $OPT(i-1, w-w_i)$ , as indicated by the arrows.

66

Problema dello zaino: approccio bottom-up

Utilizzare un array n-per-W.

$$OPT(i, w) = \begin{cases} 0 & \text{se } i=0 \\ OPT(i-1, w) & \text{se } w_i > w \\ \max\{OPT(i-1, w), v_i + OPT(i-1, w-w_i)\} & \text{altrimenti} \end{cases}$$

```

Input: n, w1, ..., wN, v1, ..., vN

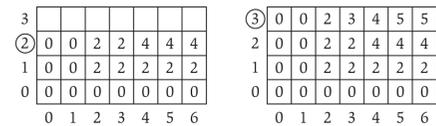
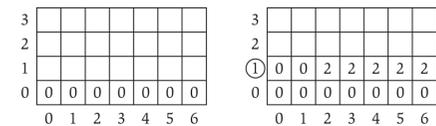
for w = 0 to W
    M[0, w] = 0

for i = 1 to n
    for w = 1 to W
        if (wi > w)
            M[i, w] = M[i-1, w]
        else
            M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi]}
    return M[n, W]
    
```

67

Problema dello zaino: esempio algoritmo

Knapsack size  $W = 6$ , items  $v_1 = 2, v_2 = 2, v_3 = 1$   
 $w_1 = 2, w_2 = 2, w_3 = 3$



**Figure 6.12** The iterations of the algorithm on a sample instance of the Subset Sum Problem.

68

### Problema dello zaino: esempio algoritmo

		W + 1											
		0	1	2	3	4	5	6	7	8	9	10	11
n + 1	$\phi$	0	0	0	0	0	0	0	0	0	0	0	0
	{1}	0	1	1	1	1	1	1	1	1	1	1	1
	{1, 2}	0	1	6	7	7	7	7	7	7	7	7	7
	{1, 2, 3}	0	1	6	7	7	18	19	24	25	25	25	25
	{1, 2, 3, 4}	0	1	6	7	7	18	22	24	28	29	29	40
	{1, 2, 3, 4, 5}	0	1	6	7	7	18	22	28	29	34	34	40

oggetto	valore	peso
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

OPT: { 4, 3 }  
valore = 22 + 18 = 40

W = 11

69

### Problema dello zaino: Complessità

Complessità.  $\Theta(nW)$ .

- Non polinomiale nella lunghezza dell' input!
- "Pseudo-polinomiale."
- Versione decisionale del problema dello zaino è NP-completo. [Cap 8]

**Algoritmi di approssimazione per lo zaino.** Esiste un algoritmo di approssimazione polinomiale che produce una soluzione il cui valore è al massimo 0.01% rispetto all'ottimale. [Sezione 11.8]

Comunque, non lo tratteremo in questo corso!

70

### Problema dello zaino: Esercizio 1

Si descriva ed analizzi un algoritmo per la seguente variazione del problema dello zaino: Dati  $n$  oggetti di peso  $w_1, w_2, \dots, w_n$  e valore  $v_1, v_2, \dots, v_n$  ed uno zaino di capacità  $W$  (tutti gli input sono  $>0$ ), trovare il massimo valore di un sottoinsieme degli oggetti il cui peso totale è al massimo  $W$ , con la condizione che ogni oggetto può essere preso anche più di una volta.

(La variazione rispetto al problema del testo, consiste nel superamento del vincolo che ogni oggetto poteva essere preso al massimo una sola volta.)

71

### Problema dello zaino: Esercizio 2

Si descriva ed analizzi un algoritmo per la seguente variazione del problema dello zaino: Dati  $n$  oggetti di peso  $w_1, w_2, \dots, w_n$  e valore  $v_1, v_2, \dots, v_n$  ed uno zaino di capacità  $W$  (tutti gli input sono  $>0$ ), trovare il massimo valore di un sottoinsieme degli oggetti il cui peso totale è al massimo  $W$ , con la condizione che ogni oggetto può essere preso al massimo 2 volte.

(La variazione rispetto al problema del testo, consiste nel superamento del vincolo che ogni oggetto poteva essere preso al massimo una sola volta.)

72

### Problema dello zaino: Esercizio 3

Si descriva ed analizzi un algoritmo per la seguente variazione del problema dello zaino: Dati  $n$  oggetti di peso  $w_1, w_2, \dots, w_n$  e valore  $v_1, v_2, \dots, v_n$  ed uno zaino di capacità  $W$  (tutti gli input sono  $>0$ ), trovare il massimo valore di un sottoinsieme degli oggetti il cui peso totale è al massimo  $W$ , con la condizione che non possono essere presi due oggetti con indici consecutivi (ovvero gli oggetti  $i$ -esimo ed  $(i+1)$ -esimo, per  $i=1, 2, \dots, n-1$ ).

73

## 6.5 RNA Secondary Structure

## Human Genome Project

Iniziato Ottobre 1990

Durata prevista 15 anni

**Scopo:** determinare la sequenza completa delle basi (3 miliardi) del DNA, identificare tutti i geni umani e renderli accessibili per ulteriori studi



75

## Genoma Umano

Fatto di DNA che ha 4 differenti mattoni chimici, chiamati basi: A, T, C, G.

Se la sequenza fosse scritta in un elenco telefonico, occorrerebbero 200 volumi di 100 pagine ognuno

Se provassimo a leggere 10 basi al secondo, cioè 600 basi/minuto, 36.000 basi/ora, 864.000 basi/giorno, 315.360.000 basi/anno, occorrerebbero circa 9,5 anni per leggere la sequenza  
1 Mb = 1.000.000 basi (megabase)

Per intera sequenza occorrono 3 gigabyte, senza contare altri dati associati

76

## Genoma Umano

Human Genome Research Institute (finanziamenti pubblici USA) e Celera Genomics Corporation (settore commerciale)

Sequenze geniche sono brevettabili, stabilito dai tribunali (brevetto dura 20 anni)

Corsa ai brevetti per Celera ed alla pubblicazione degli accademici per evitare i brevetti

Corsa finita alla pari

26 giugno 2000, Tony Blair e Bill Clinton annunciano il sequenziamento completo del genoma umano

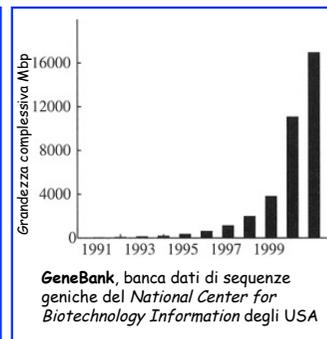
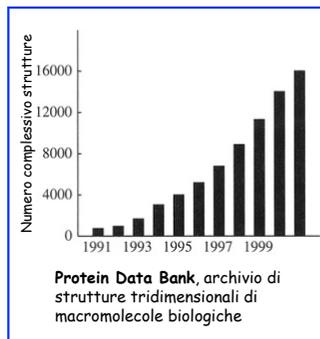


77

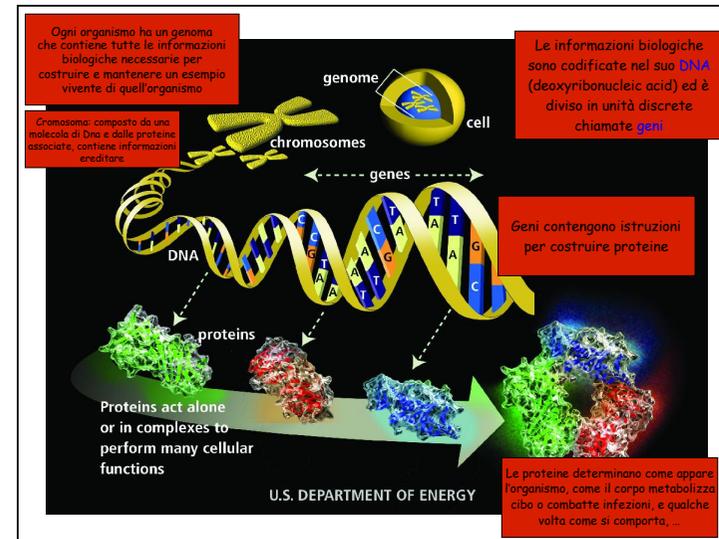
Organism	Genome Size (Bases)	Estimated Genes
Human ( <i>Homo sapiens</i> )	3 billion	30,000
Laboratory mouse ( <i>M. musculus</i> )	2.6 billion	30,000
Mustard weed ( <i>A. thaliana</i> )	100 million	25,000
Roundworm ( <i>C. elegans</i> )	97 million	19,000
Fruit fly ( <i>D. melanogaster</i> )	137 million	13,000
Yeast ( <i>S. cerevisiae</i> )	12.1 million	6,000
Bacterium ( <i>E. coli</i> )	4.6 million	3,200
Human immunodeficiency virus (HIV)	9700	9

78

## Crescita Banche Dati



79





### Ribonucleic acid (RNA)

Simile al DNA.

Singola catena con 4 nucleotidi: adenine (A), cytosine (C), guanine (G), uracil (U).

RNA. Stringa  $B = b_1b_2\dots b_n$  su alfabeto  $\{A, C, G, U\}$ .

85

### RNA Secondary Structure

RNA. Stringa  $B = b_1b_2\dots b_n$  su alfabeto  $\{A, C, G, U\}$ .

**Struttura secondaria.** RNA è una singola catena e tende a formare coppie di basi con se stessa. Questa struttura è essenziale per capire il comportamento delle molecole.

coppie di base complementari: A-U, C-G

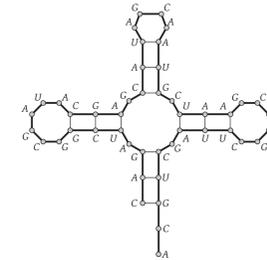


Figure 6.13 An RNA secondary structure. Thick lines connect adjacent elements of the sequence; thin lines indicate pairs of elements that are matched.

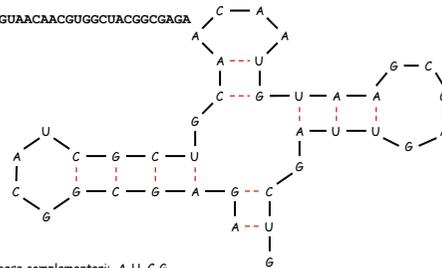
86

### RNA Secondary Structure

RNA. Stringa  $B = b_1b_2\dots b_n$  su alfabeto  $\{A, C, G, U\}$ .

**Struttura secondaria.** RNA è una singola catena e tende a formare coppie di basi con se stessa. Questa struttura è essenziale per capire il comportamento delle molecole.

Esempio: GUCGAUUGAGCGAAUGUAACAACGUGGCUAACGCGGAGA



coppie di base complementari: A-U, C-G

87

### RNA Secondary Structure

**Struttura secondaria.** Un insieme di coppie  $S = \{(b_i, b_j)\}$  che soddisfa:

- [Watson-Crick.]  $S$  è un matching ed ogni coppia in  $S$  è un complemento Watson-Crick: A-U, U-A, C-G, oppure G-C.

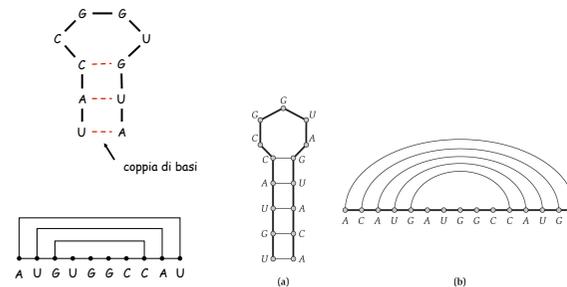


Figure 6.14 Two views of an RNA secondary structure. In the second view, (b), the string has been "stretched" lengthwise, and edges connecting matched pairs appear as noncrossing "bubbles" over the string.

88

RNA Secondary Structure

**Struttura secondaria.** Un insieme di coppie  $S = \{(b_i, b_j)\}$  che soddisfa:

- [Watson-Crick.]  $S$  è un matching ed ogni coppia in  $S$  è un complemento Watson-Crick: A-U, U-A, C-G, oppure G-C.
- [No sharp turns.] Gli elementi di ogni coppia sono separati da almeno 4 basi. Se  $(b_i, b_j) \in S$ , allora  $i < j - 4$ .

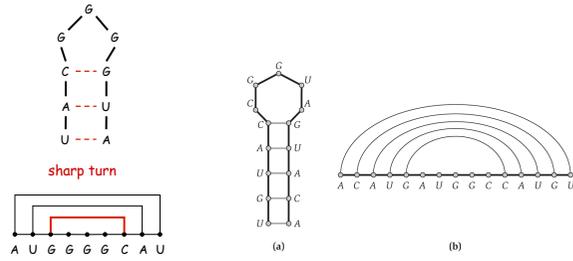


Figure 6.14 Two views of an RNA secondary structure. In the second view, (b), the string has been "stretched" lengthwise, and edges connecting matched pairs appear as noncrossing "bubbles" over the string.

89

RNA Secondary Structure

**Struttura secondaria.** Un insieme di coppie  $S = \{(b_i, b_j)\}$  che soddisfa:

- [Watson-Crick.]  $S$  è un matching ed ogni coppia in  $S$  è un complemento Watson-Crick: A-U, U-A, C-G, oppure G-C.
- [No sharp turns.] Gli elementi di ogni coppia sono separati da almeno 4 basi. Se  $(b_i, b_j) \in S$ , allora  $i < j - 4$ .
- [Non-crossing.] Se  $(b_i, b_j)$  e  $(b_k, b_l)$  sono due coppie in  $S$ , allora non è possibile che  $i < k < j < l$ .

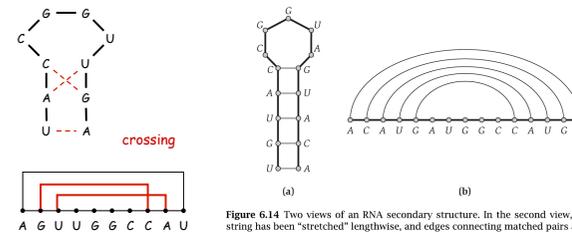


Figure 6.14 Two views of an RNA secondary structure. In the second view, (b), the string has been "stretched" lengthwise, and edges connecting matched pairs appear as noncrossing "bubbles" over the string.

90

RNA Secondary Structure

**Struttura secondaria.** Un insieme di coppie  $S = \{(b_i, b_j)\}$  che soddisfa:

- [Watson-Crick.]  $S$  è un matching ed ogni coppia in  $S$  è un complemento Watson-Crick: A-U, U-A, C-G, oppure G-C.
- [No sharp turns.] Gli elementi di ogni coppia sono separati da almeno 4 basi. Se  $(b_i, b_j) \in S$ , allora  $i < j - 4$ .
- [Non-crossing.] Se  $(b_i, b_j)$  e  $(b_k, b_l)$  sono due coppie in  $S$ , allora non è possibile che  $i < k < j < l$ .

**Energia libera.** L'ipotesi usuale è che una molecola RNA formerà la struttura secondaria con l'energia libera totale ottimale.

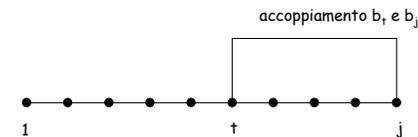
↙  
approssimativamente il numero di coppie di basi

**Obiettivo.** Data una molecola RNA  $B = b_1b_2...b_n$ , trovare una struttura secondaria  $S$  che massimizza il numero di coppie di basi.

91

RNA Secondary Structure: Sottoproblemi

**Primo tentativo.**  $OPT(j)$  = massimo numero di coppie di basi in una struttura secondaria della sottostringa  $b_1b_2...b_j$ .



**Difficoltà.** Abbiamo due sottoproblemi:

- Trovare la struttura secondaria in:  $b_1b_2...b_{t-1}$ . ←  $OPT(t-1)$
- Trovare la struttura secondaria in:  $b_{t+1}b_{t+2}...b_{j-1}$ . ← abbiamo bisogno di più sottoproblemi

92

Programmazione dinamica su intervalli

**Notazione.**  $OPT(i, j)$  = massimo numero di coppie di basi in una struttura secondaria della sottostringa  $b_i b_{i+1} \dots b_j$ .

- Case 1. Se  $i \geq j - 4$ .  
-  $OPT(i, j) = 0$  per la condizione "no-sharp turns".
- Case 2. La base  $b_j$  non fa parte di una coppia.  
-  $OPT(i, j) = OPT(i, j-1)$
- Case 3. La base  $b_j$  fa coppia con  $b_t$  per qualche  $i \leq t < j - 4$ .  
- Vincolo "non-crossing" determina i risultanti sottoproblemi  
-  $OPT(i, j) = \max_{i \leq t < j-4} \{ 1 + OPT(i, t-1) + OPT(t+1, j-1) \}$   
 $b_t$  e  $b_j$  sono complementi Watson-Crick, cioè A-U, U-A, C-G, oppure G-C.

93

Programmazione dinamica su intervalli

**Notazione.**  $OPT(i, j)$  = massimo numero di coppie di basi in una struttura secondaria della sottostringa  $b_i b_{i+1} \dots b_j$ .

- Case 1. Se  $i \geq j - 4$ .  
-  $OPT(i, j) = 0$  per la condizione "no-sharp turns".
- Case 2. La base  $b_j$  non fa parte di una coppia.  
-  $OPT(i, j) = OPT(i, j-1)$
- Case 3. La base  $b_j$  fa coppia con  $b_t$  per qualche  $i \leq t < j - 4$ .  
- Vincolo "non-crossing" determina i risultanti sottoproblemi  
-  $OPT(i, j) = \max_{i \leq t < j-4} \{ 1 + OPT(i, t-1) + OPT(t+1, j-1) \}$   
 $b_t$  e  $b_j$  sono complementi Watson-Crick, cioè A-U, U-A, C-G, oppure G-C.

$$OPT(i, j) = \begin{cases} 0 & i \geq j - 4 \\ \max \left\{ \begin{array}{l} OPT(i, j-1) \\ \max_{\substack{t < j-4 \\ b_t, b_j \text{ complementi}}} \{ 1 + OPT(i, t-1) + OPT(t+1, j-1) \} \end{array} \right\} & \text{altrimenti} \end{cases}$$

94

Bottom Up Dynamic Programming Over Intervals

**Domanda.** In che ordine risolvere i sottoproblemi?  
**Risposta.** Dapprima gli intervalli più corti.

```

RNA (b1, ..., bn) {
  for k = 5, 6, ..., n-1
    for i = 1, 2, ..., n-k
      j = i + k
      Compute M[i, j]
  return M[1, n]
}
        
```

k è la lunghezza intervallo j-i

usare ricorrenza

$$OPT(i, j) = \max \begin{cases} OPT(i, j-1) \\ \max_{\substack{t < j-4 \\ b_t, b_j \text{ complementi}}} \{ 1 + OPT(i, t-1) + OPT(t+1, j-1) \} \end{cases}$$

**Complessità.**  $O(n^3)$ .

95

RNA sequence ACCGGUAGU

4	0	0	0	0
3	0	0		
2	0			
i = 1				
	j = 6	7	8	9

Initial values

4	0	0	0	0
3	0	0	1	
2	0	0		
i = 1	1			
	j = 6	7	8	9

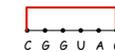
Filling in the values for k = 5

$$OPT(i, j) = \max \begin{cases} OPT(i, j-1) \\ \max_{\substack{t < j-4 \\ b_t, b_j \text{ complementi}}} \{ 1 + OPT(i, t-1) + OPT(t+1, j-1) \} \end{cases}$$

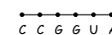
$$OPT(4, 9) = \max_{\substack{t=4 \\ b_t, b_9 \text{ complementi}}} \{ 1 + OPT(4, t-1) + OPT(t+1, 8) \} = 0$$



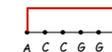
$$OPT(3, 8) = \max_{\substack{t=4 \\ b_t, b_8 \text{ complementi}}} \{ 1 + OPT(1, t-1) + OPT(t+1, 7) \} = 1$$



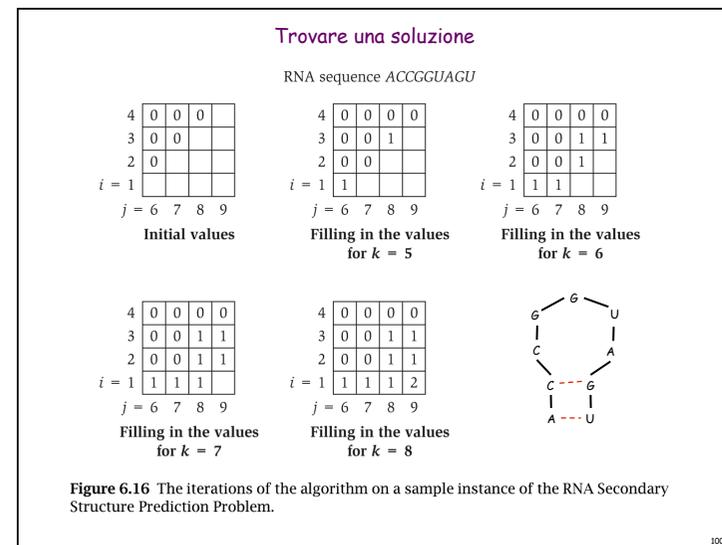
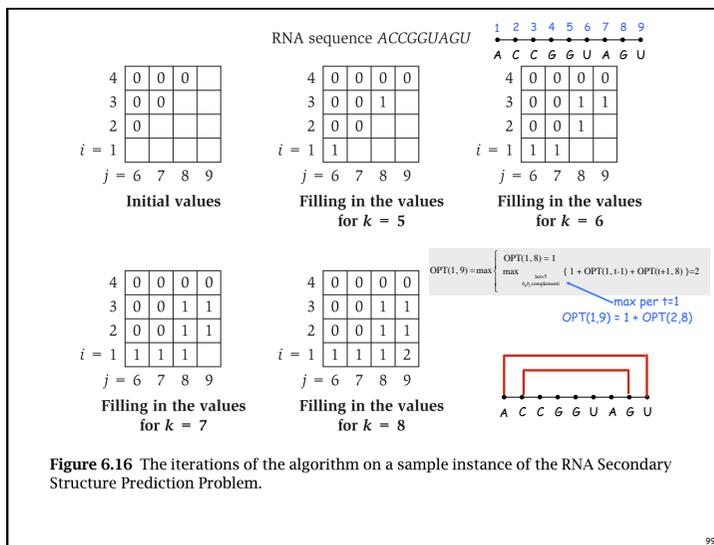
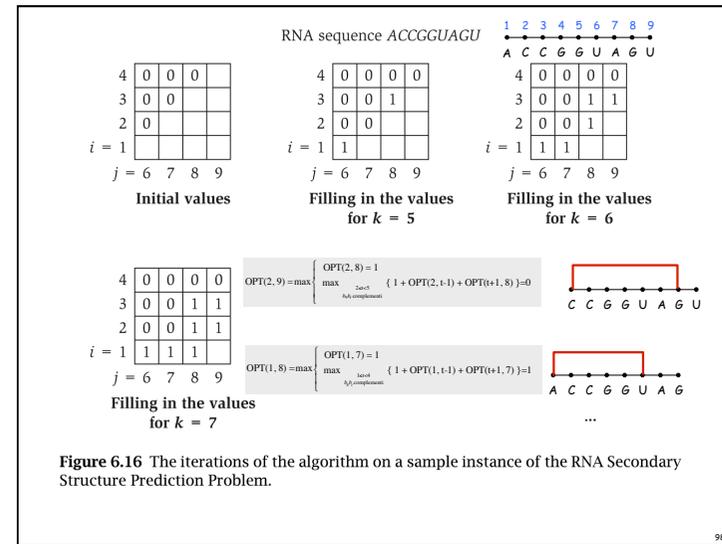
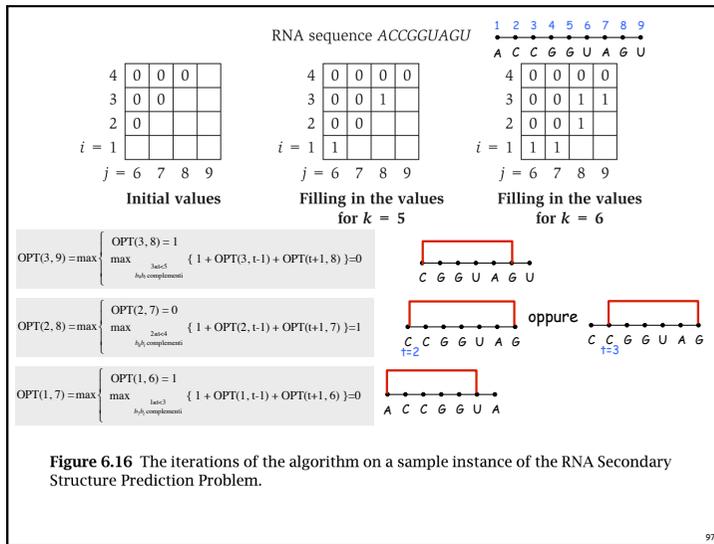
$$OPT(2, 7) = \max_{\substack{t=3 \\ b_t, b_7 \text{ complementi}}} \{ 1 + OPT(2, t-1) + OPT(t+1, 6) \} = 0$$



$$OPT(1, 6) = \max_{\substack{t=2 \\ b_t, b_6 \text{ complementi}}} \{ 1 + OPT(1, t-1) + OPT(t+1, 5) \} = 1$$



96



### Programmazione dinamica: sommario

#### Ricetta.

- Caratterizza struttura del problema.
- Definire ricorsivamente valore della soluzione ottimale.
- Computare valore soluzione ottimale.
- Costruire soluzione ottimale dalle informazioni computate.

#### Tecniche di Programmazione dinamica.

- Scelta binaria: weighted interval scheduling.
- Multi-way choice: segmented least squares.
- Aggiunta di una nuova variabile: knapsack.
- Programmazione dinamica su intervalli: RNA secondary structure.

Top-down vs. bottom-up: persone diverse hanno diverse intuizioni.

101

## 6.6 Sequence Alignment

### Similarità di stringhe

#### Quanto sono simili due stringhe?

- occurrence
- occurrence

o c u r r a n c e -  
o c c u r r e n c e

6 mismatches, 1 gap

o c - u r r a n c e  
o c c u r r e n c e

1 mismatch, 1 gap

o c - u r r - a n c e  
o c c u r r e - n c e

0 mismatches, 3 gaps

103

### Edit Distance

#### Applicazioni.

- Base per diff di Unix.
- Riconoscimento voce.
- Biologia Computazionale.

#### Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]

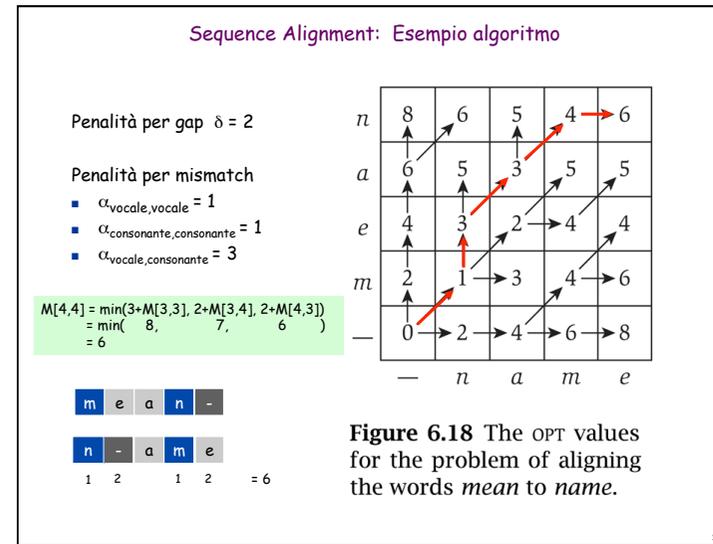
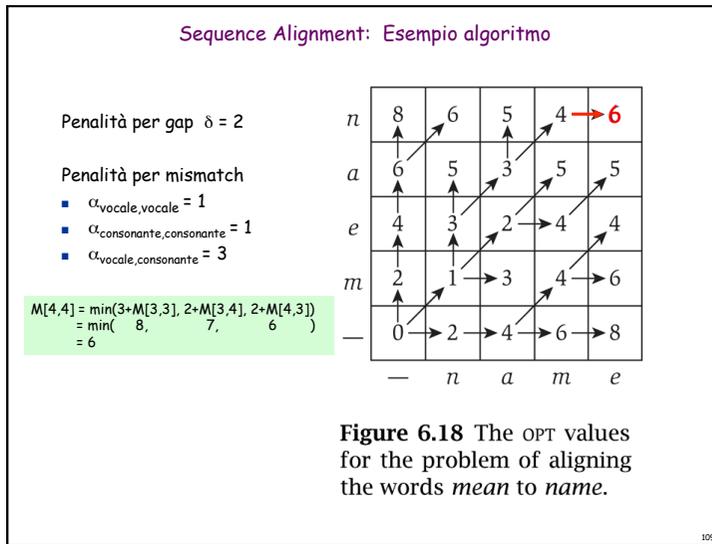
- Penalità per gap  $\delta$ ; penalità per mismatch  $\alpha_{pq}$ .
- Costo = somma delle penalità di gap e mismatch.

C T G A C C T A C C T    - C T G A C C T A C C T  
C C T G A C T A C A T    C C T G A C - T A C A T

$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$                        $2\delta + \alpha_{CA}$

104





### Sequence Alignment: Complessità algoritmo

```

Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α) {
  for i = 0 to m
    M[0, i] = iδ
  for j = 0 to n
    M[j, 0] = jδ

  for i = 1 to m
    for j = 1 to n
      M[i, j] = min(α[xi, yj] + M[i-1, j-1],
                  δ + M[i-1, j],
                  δ + M[i, j-1])

  return M[m, n]
}
    
```

**Analisi.**  $\Theta(mn)$  tempo e spazio.

**Parole o frasi in italiano o inglese:**  $m, n \leq 10$ .

**Biologia Computazionale:**  $m = n = 100.000$ .

Va bene per 10 miliardi operazioni, ma array 10GB?

### Sequence Alignment: Linear Space

**Domanda.** Possiamo usare meno spazio?

**Facile.** Valore ottimo in spazio  $O(m+n)$  e tempo  $O(mn)$ .

- Computare  $OPT(i, \cdot)$  da  $OPT(i-1, \cdot)$ .
- Non è più facile computare un allineamento ottimale.

**Teorema.** [Hirschberg 1975] Allineamento ottimo in spazio  $O(m+n)$  e tempo  $O(mn)$ .

- Combinazione di divide-and-conquer e programmazione dinamica.

### 6.7 Sequence Alignment in Linear Space

... ma non lo tratteremo

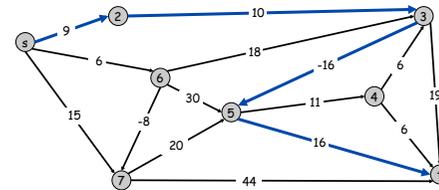
## 6.8 Shortest Paths

### Shortest Paths

**Problema del cammino più corto.** Dato un grafo diretto  $G = (V, E)$ , con peso degli archi  $c_{vw}$ , trovare il cammino più corto dal nodo  $s$  al nodo  $t$ .

anche pesi negativi

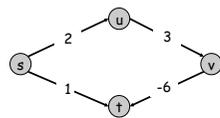
**Esempio.** I nodi rappresentano agenti in ambito finanziario  $c_{vw}$  è il costo di una transazione nella quale si acquista da  $v$  e si vende a  $w$ . Un cammino rappresenta una successione di transazioni.



114

### Shortest Paths: Approcci che non funzionano

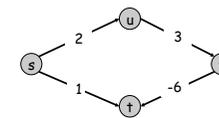
**Algoritmo di Dijkstra.** Potrebbe non funzionare con costi negativi.



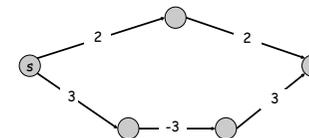
115

### Shortest Paths: Approcci che non funzionano

**Algoritmo di Dijkstra.** Potrebbe non funzionare con costi negativi.



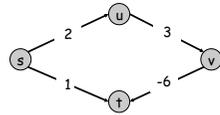
**Scalare il peso degli archi.** Aggiungere una costante ad ogni peso degli archi potrebbe non funzionare.



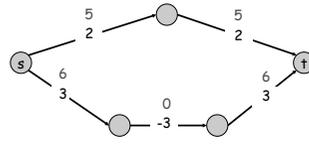
116

Shortest Paths: Approcci che non funzionano

Algoritmo di Dijkstra. Potrebbe non funzionare con costi negativi.



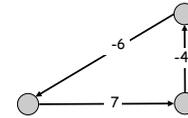
Scalare il peso degli archi. Aggiungere una costante ad ogni peso degli archi potrebbe non funzionare.



117

Shortest Paths: Ciclo con costo negativo

Ciclo con costo negativo.



Osservazione. Se un cammino da s a t contiene un ciclo con costo negativo, non esiste un cammino s-t più corto; comunque, ne esiste uno che è semplice.

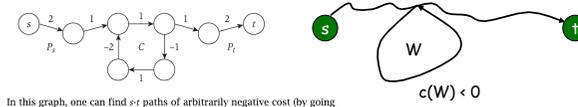


Figure 6.20 In this graph, one can find s-t paths of arbitrarily negative cost (by going around the cycle C many times).

Assumiamo che il grafo non contenga cicli con costo negativo!

118

Shortest Paths: Programmazione dinamica

Definizione.  $OPT(i, v)$  = lunghezza del cammino P più corto v-t usando al massimo i archi.

- Caso 1: P usa al massimo i-1 archi.
  - $OPT(i, v) = OPT(i-1, v)$
- Caso 2: P usa esattamente i archi.
  - se  $(v, w)$  è il primo arco, allora OPT usa  $(v, w)$ , e poi sceglie il miglior cammino w-t usando al massimo i-1 archi
  - $OPT(i, v) = OPT(i-1, w) + c_{vw}$

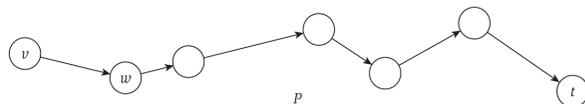


Figure 6.22 The minimum-cost path P from v to t using at most i edges.

119

Shortest Paths: Programmazione dinamica

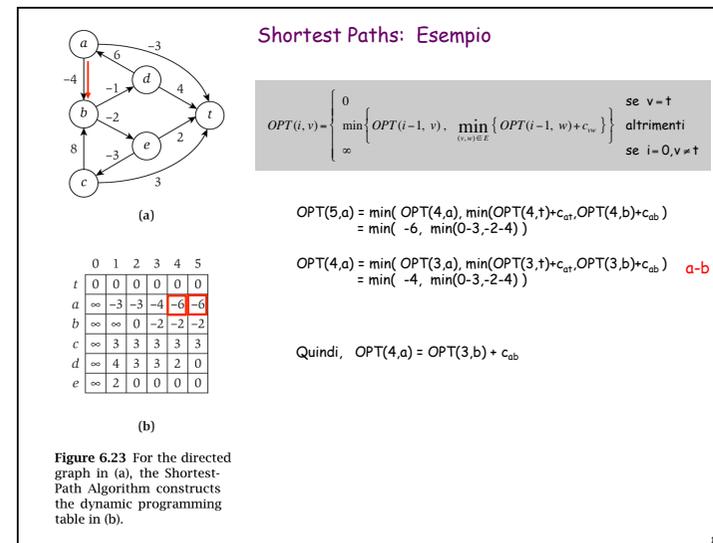
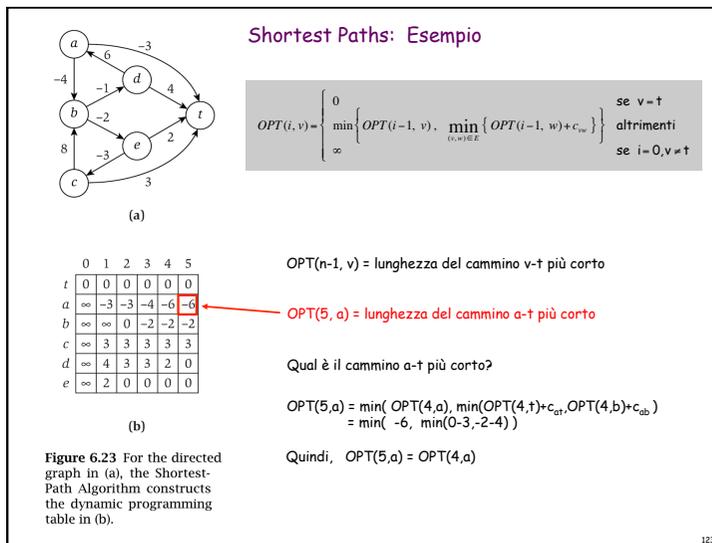
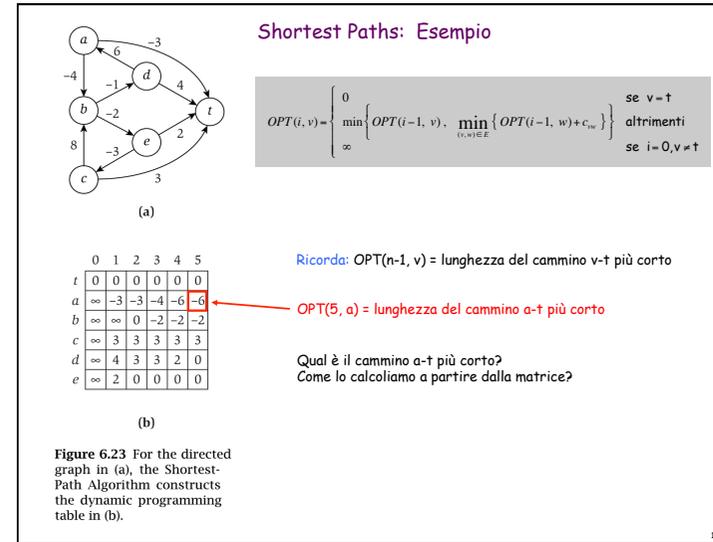
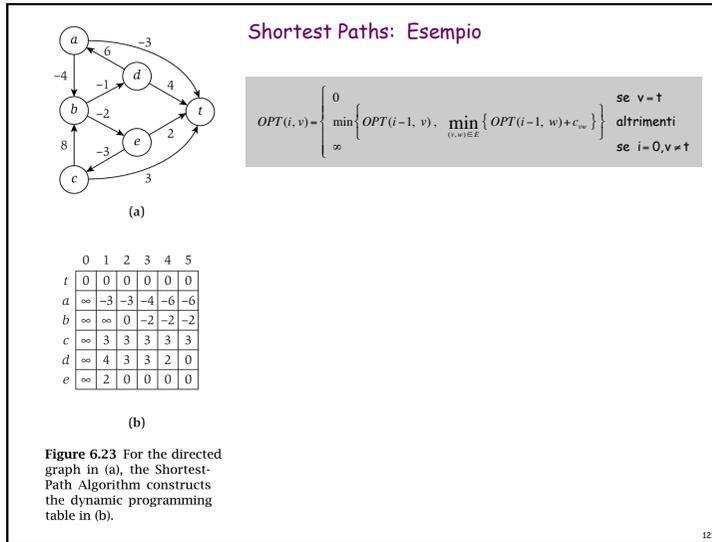
Definizione.  $OPT(i, v)$  = lunghezza del cammino P più corto v-t usando al massimo i archi.

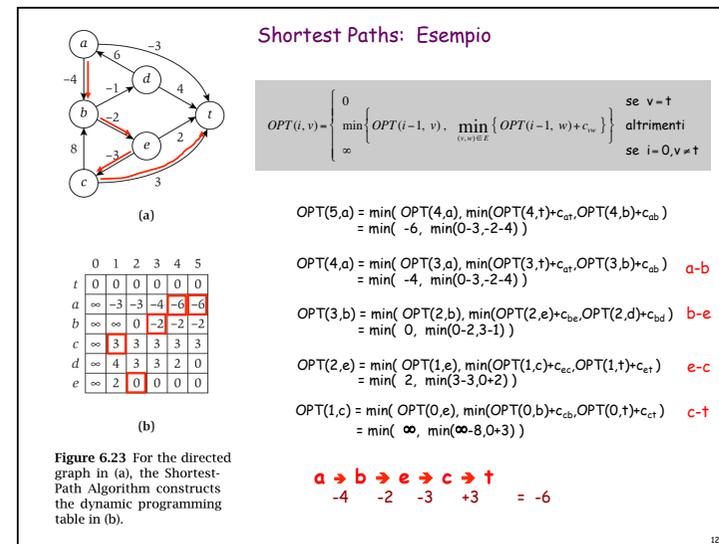
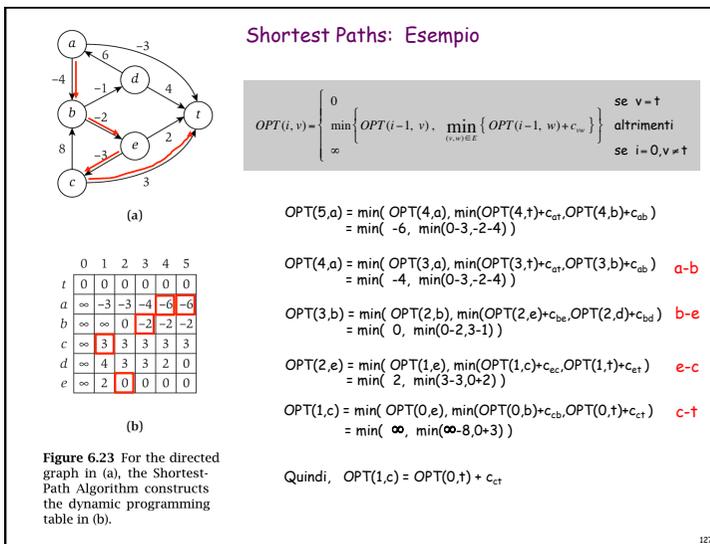
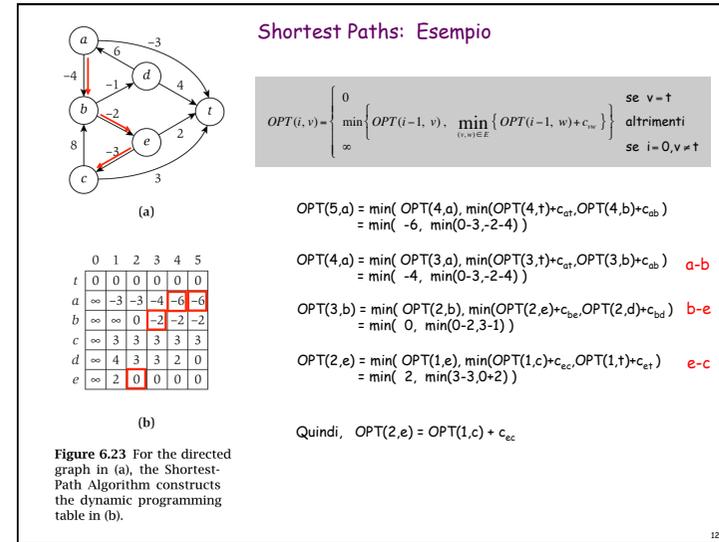
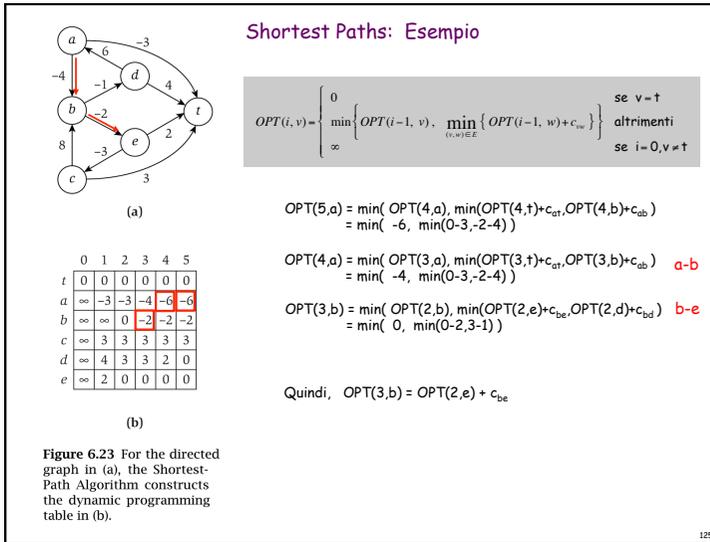
- Caso 1: P usa al massimo i-1 archi.
  - $OPT(i, v) = OPT(i-1, v)$
- Caso 2: P usa esattamente i archi.
  - se  $(v, w)$  è il primo arco, allora OPT usa  $(v, w)$ , e poi sceglie il miglior cammino w-t usando al massimo i-1 archi

$$OPT(i, v) = \begin{cases} 0 & \text{se } v = t \\ \min \left\{ OPT(i-1, v), \min_{(v,w) \in E} \{ OPT(i-1, w) + c_{vw} \} \right\} & \text{altrimenti} \\ \infty & \text{se } i = 0, v \neq t \end{cases}$$

Nota.  $OPT(n-1, v)$  = lunghezza del cammino v-t più corto (Ricorda che non ci sono cicli con costo negativo)

120





### Shortest Paths: Esempio

$$OPT(i, v) = \begin{cases} 0 & \text{se } v = t \\ \min \left\{ OPT(i-1, v), \min_{(i,w) \in E} \{ OPT(i-1, w) + c_{iw} \} \right\} & \text{altrimenti} \\ \infty & \text{se } i = 0, v \neq t \end{cases}$$

$$OPT(5, a) = \min( OPT(4, a), \min(OPT(4, t) + c_{at}, OPT(4, b) + c_{ab}) ) = \min( -6, \min(0-3, -2-4) )$$

$$OPT(4, a) = \min( OPT(3, a), \min(OPT(3, t) + c_{at}, OPT(3, b) + c_{ab}) ) = \min( -4, \min(0-3, -2-4) )$$
 **a-b**

$$OPT(3, b) = \min( OPT(2, b), \min(OPT(2, e) + c_{be}, OPT(2, d) + c_{bd}) ) = \min( 0, \min(0-2, 3-1) )$$
 **b-e**

$$OPT(2, e) = \min( OPT(1, e), \min(OPT(1, c) + c_{ec}, OPT(1, t) + c_{et}) ) = \min( 2, \min(3-3, 0+2) )$$
 **e-c**

$$OPT(1, c) = \min( OPT(0, e), \min(OPT(0, b) + c_{cb}, OPT(0, t) + c_{ct}) ) = \min( \infty, \min(\infty-8, 0+3) )$$
 **c-t**

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	$\infty$	-3	-3	-4	-6	-6
b	$\infty$	$\infty$	0	-2	-2	-2
c	$\infty$	3	3	3	3	3
d	$\infty$	4	3	3	2	0
e	$\infty$	2	0	0	0	0

Figure 6.23 For the directed graph in (a), the Shortest-Path Algorithm constructs the dynamic programming table in (b).

### Shortest Paths: Esempio

Trovare il cammino più corto.  
Qual'è l'informazione che possiamo ricordare?

$$OPT(5, a) = \min( OPT(4, a), \min(OPT(4, t) + c_{at}, OPT(4, b) + c_{ab}) ) = \min( -6, \min(0-3, -2-4) )$$

$$OPT(4, a) = \min( OPT(3, a), \min(OPT(3, t) + c_{at}, OPT(3, b) + c_{ab}) ) = \min( -4, \min(0-3, -2-4) )$$
 **a-b**

$$OPT(3, b) = \min( OPT(2, b), \min(OPT(2, e) + c_{be}, OPT(2, d) + c_{bd}) ) = \min( 0, \min(0-2, 3-1) )$$
 **b-e**

$$OPT(2, e) = \min( OPT(1, e), \min(OPT(1, c) + c_{ec}, OPT(1, t) + c_{et}) ) = \min( 2, \min(3-3, 0+2) )$$
 **e-c**

$$OPT(1, c) = \min( OPT(0, e), \min(OPT(0, b) + c_{cb}, OPT(0, t) + c_{ct}) ) = \min( \infty, \min(\infty-8, 0+3) )$$
 **c-t**

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	$\infty$	-3	-3	-4	-6	-6
b	$\infty$	$\infty$	0	-2	-2	-2
c	$\infty$	3	3	3	3	3
d	$\infty$	4	3	3	2	0
e	$\infty$	2	0	0	0	0

Figure 6.23 For the directed graph in (a), the Shortest-Path Algorithm constructs the dynamic programming table in (b).

### Shortest Paths: Esempio

Trovare il cammino più corto.  
Qual'è l'informazione che possiamo ricordare?  
Mantenere un "successore" per ogni valore della tabella.

$$OPT(5, a) = \min( OPT(4, a), \min(OPT(4, t) + c_{at}, OPT(4, b) + c_{ab}) ) = \min( -6, \min(0-3, -2-4) )$$

$$OPT(4, a) = \min( OPT(3, a), \min(OPT(3, t) + c_{at}, OPT(3, b) + c_{ab}) ) = \min( -4, \min(0-3, -2-4) )$$
 **a-b**

$$OPT(3, b) = \min( OPT(2, b), \min(OPT(2, e) + c_{be}, OPT(2, d) + c_{bd}) ) = \min( 0, \min(0-2, 3-1) )$$
 **b-e**

$$OPT(2, e) = \min( OPT(1, e), \min(OPT(1, c) + c_{ec}, OPT(1, t) + c_{et}) ) = \min( 2, \min(3-3, 0+2) )$$
 **e-c**

$$OPT(1, c) = \min( OPT(0, e), \min(OPT(0, b) + c_{cb}, OPT(0, t) + c_{ct}) ) = \min( \infty, \min(\infty-8, 0+3) )$$
 **c-t**

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	$\infty$	-3	-3	-4	-6	-6
b	$\infty$	$\infty$	0	-2	-2	-2
c	$\infty$	3	3	3	3	3
d	$\infty$	4	3	3	2	0
e	$\infty$	2	0	0	0	0

Figure 6.23 For the directed graph in (a), the Shortest-Path Algorithm constructs the dynamic programming table in (b).

### Shortest Paths: Esercizio

Si esegua l'algoritmo di programmazione dinamica Shortest-Path(G, t) per il calcolo dei cammini minimi sul grafo G

Si chiariscano i passi effettuati evidenziando i valori della matrice OPT costruita dall'algoritmo

	0	1	2	3
t	0	0	0	0
u	$\infty$			
v	$\infty$			
s	$\infty$			

Si descriva come ottenere il cammino di costo minimo dal nodo u facendo uso della matrice OPT e chiarendo i passi effettuati.

## Shortest Paths: Implementazione

```

Shortest-Path(G, t) {
  array M[0..n-1, V]
  foreach node v ∈ V
    M[0, v] ← ∞
  M[0, t] ← 0

  for i = 1 to n-1
    foreach node v ∈ V
      M[i, v] ← M[i-1, v]
    foreach edge (v, w) ∈ E
      M[i, v] ← min { M[i, v], M[i-1, w] + cvw }
}

```

133

## Shortest Paths: Implementazione

```

Shortest-Path(G, t) {
  array M[0..n-1, V]
  foreach node v ∈ V
    M[0, v] ← ∞
  M[0, t] ← 0

  for i = 1 to n-1
    foreach node v ∈ V
      M[i, v] ← M[i-1, v]
    foreach edge (v, w) ∈ E
      M[i, v] ← min { M[i, v], M[i-1, w] + cvw }
}

```

Analisi. Tempo  $\Theta(mn)$ , spazio  $\Theta(n^2)$ .

134

## Shortest Paths: Miglioramenti pratici

## Miglioramenti pratici.

- Mantenere solo un array  $M[v]$  = cammino più corto v-t che abbiamo trovato finora. ("i" è solo un contatore)

$$M(i, v) = \min \left\{ M(i-1, v), \min_{(v,w) \in E} \{ M(i-1, w) + c_{vw} \} \right\}$$

- la relazione "si semplifica":  $M(v) = \min \left\{ M(v), \min_{(v,w) \in E} \{ M(w) + c_{vw} \} \right\}$

```

Shortest-Path(G, t) {
  array M[V]
  foreach node v ∈ V
    M[v] ← ∞
  M[t] ← 0

  for i = 1 to n-1
    foreach edge (v, w) ∈ E
      M[v] ← min { M[v], M[w] + cvw }
}

```

135

## Shortest Paths: Miglioramenti pratici

## Miglioramenti pratici.

- Mantenere solo un array  $M[v]$  = cammino più corto v-t che abbiamo trovato finora.
- Nessuna necessità di controllare archi della forma (v, w) a meno che  $M[w]$  è cambiato nell'iterazione precedente.

**Teorema.** Durante l'algoritmo,  $M[v]$  è la lunghezza di un cammino v-t, e dopo i round di aggiornamenti, il valore  $M[v]$  non è maggiore della lunghezza del cammino v-t più corto usando  $\leq i$  archi.

## Impatto totale.

- Memoria:  $O(m + n)$ .
- Complessità tempo: caso peggiore  $O(mn)$ , ma veloce in pratica.

136

Esercizio: Longest Common Subsequence

Problema: Date 2 sequenze,  $X = x_1 \dots x_m$  e  $Y = y_1 \dots y_n$ , trovare una sottosequenza comune la cui lunghezza è massima.



Algoritmo naïve:  $\Theta(n2^m)$   
(per ogni sottosequenza di X, verificare se è anche sottosequenza di Y)

137

Esercizio: Longest Common Subsequence

Problema: Date 2 sequenze,  $X = x_1 \dots x_m$  e  $Y = y_1 \dots y_n$ , trovare una sottosequenza comune la cui lunghezza è massima.



Algoritmo naïve:  $\Theta(n2^m)$   
(per ogni sottosequenza di X, verificare se è anche sottosequenza di Y)

Descrivere ed analizzare un algoritmo di programmazione dinamica  
Suggerimento: la tecnica è simile a quella per Sequence Alignment

138

Longest Common Subsequence: sottostruttura ottimale

- Sia  $Z = z_1 \dots z_k$  una LCS di X e Y
1. Se  $x_m = y_n$ , allora  $z_k = x_m = y_n$  e  $Z_{k-1}$  è una LCS di  $X_{m-1}$  e  $Y_{n-1}$
  2. Se  $x_m \neq y_n$ , allora
    - $z_k \neq x_m$  e Z è una LCS di  $X_{m-1}$  e Y oppure
    - $z_k \neq y_n$  e Z è una LCS di X e  $Y_{n-1}$

Notazione:  
 $X_i = x_1 \dots x_i$  è un prefisso di X  
 $Y_i = y_1 \dots y_i$  è un prefisso di Y

139

Longest Common Subsequence: soluzione ricorsiva

- Sia  $Z = z_1 \dots z_k$  una LCS di X e Y
1. Se  $x_m = y_n$ , allora  $z_k = x_m = y_n$  e  $Z_{k-1}$  è una LCS di  $X_{m-1}$  e  $Y_{n-1}$
  2. Se  $x_m \neq y_n$ , allora
    - $z_k \neq x_m$  e Z è una LCS di  $X_{m-1}$  e Y oppure
    - $z_k \neq y_n$  e Z è una LCS di X e  $Y_{n-1}$

Definiamo  $c[i,j]$  = lunghezza della LCS di  $X_i$  e  $Y_j$   
La soluzione del problema è data da  $c[m,n]$ .

$$c[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ oppure } j = 0, \\ c[i-1,j-1]+1 & \text{se } i, j > 0 \text{ e } x_i = y_j, \\ \max(c[i-1,j], c[i,j-1]) & \text{se } i, j > 0 \text{ e } x_i \neq y_j. \end{cases}$$

140

Longest Common Subsequence: algoritmo iterativo

```

LCS (X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
  do c[i,0] ← 0
for j ← 0 to n
  do c[0,j] ← 0
for i ← 1 to m
  do for j ← 1 to n
    do if xi = yj
      then c[i,j] ← c[i-1,j-1] + 1
      b[i,j] ← "\
    else if c[i-1,j] ≥ c[i,j-1]
      then c[i,j] ← c[i-1,j]
      b[i,j] ← "↑"
    else c[i,j] ← c[i,j-1]
      b[i,j] ← "←"
return c and b
    
```

b[i,j] è un puntatore al sottoproblema usato per risolvere la LCS di X<sub>i</sub> e Y<sub>j</sub>

I valori b[i,j] sono utili per calcolare la sottosequenza di lunghezza massima

Il valore c[m,n] è la lunghezza della LCS di X e Y.

Complessità O(mn)

141

Longest Common Subsequence: esempio

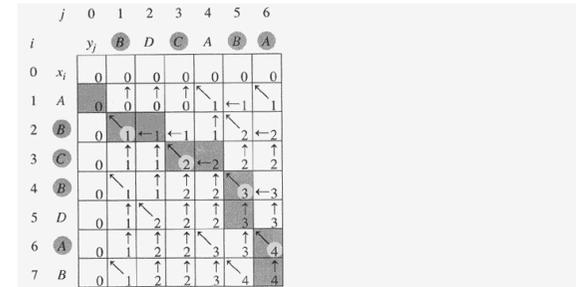


Figure 15.6 The c and b tables computed by LCS-LENGTH on the sequences X = (A, B, C, B, D, A, B) and Y = (B, D, C, A, B, A). The square in row i and column j contains the value of c[i, j] and the appropriate arrow for the value of b[i, j]. The entry 4 in c[7, 6]—the lower right-hand corner of the table—is the length of an LCS (B, C, B, A) of X and Y. For i, j > 0, entry c[i, j] depends only on whether x<sub>i</sub> = y<sub>j</sub> and the values in entries c[i - 1, j], c[i, j - 1], and c[i - 1, j - 1], which are computed before c[i, j]. To reconstruct the elements of an LCS, follow the b[i, j] arrows from the lower right-hand corner; the path is shaded. Each “\,” on the path corresponds to an entry (highlighted) for which x<sub>i</sub> = y<sub>j</sub> is a member of an LCS.

142

Longest Common Subsequence: costruzione di una LCS

```

PRINT-LCS (b, X, i, j)
if i = 0 or j = 0
  then return
if b[i, j] = "\
  then PRINT-LCS(b, X, i-1, j-1)
  print xi
  else if b[i, j] = "↑"
    then PRINT-LCS(b, X, i-1, j)
  else PRINT-LCS(b, X, i, j-1)
    
```

- Chiamata iniziale: PRINT-LCS (b, X, i, j)
- Quando b[i, j] = “\”, la LCS viene estesa di un carattere.
- Running Time: O(m + n)

143

Riepilogo Cap. 6, Dynamic Programming

- 6.1 Weighted Interval Scheduling
- 6.2 Principles of Dynamic Programming
- 6.3 Segmented Least Squares
- 6.4 Knapsack Problem
- 6.5 RNA Secondary Structure
- 6.6 Sequence Alignment
- 6.7 Sequence Alignment in Linear Space (no)
- 6.8 Shortest Paths in a Graph
- 6.9 Shortest Paths and Distance Vector Protocols (no)
- 6.10 Negative Cycles in a Graph (no)

- Esercizio:
- Longest Common Subsequence

144