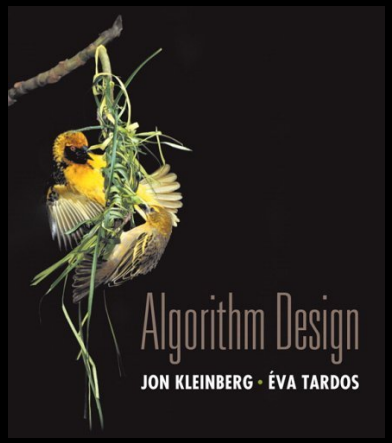


Chapter 6

Dynamic Programming

PEARSON Addison Wesley

Paradigmi algoritmici

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

Divide-and-conquer. Divide un problema in due (o più) 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.

Programmazione Dinamica: storia

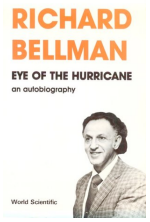
Richard Bellman. E' stato il pioniere dello studio sistematico della programmazione dinamica negli anni '50.

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"



Programmazione Dinamica: Applicazioni

Aree.

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

Alcni algoritmi di programmazione dinamica famosi.

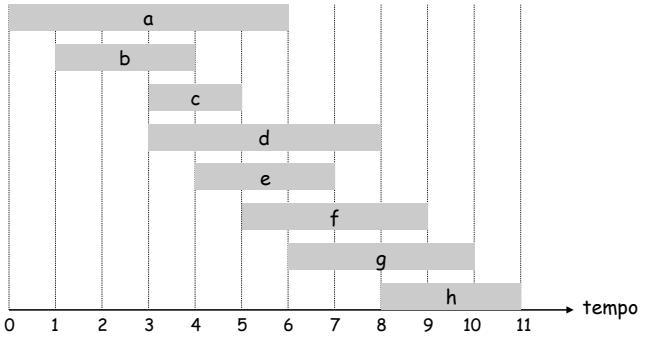
- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- Smith-Waterman for sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

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 compatili se non hanno intersezione.
- Obiettivo: trovare sottoinsieme con massimo peso di job mutuamente compatibili.



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.

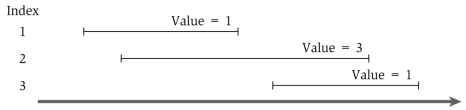
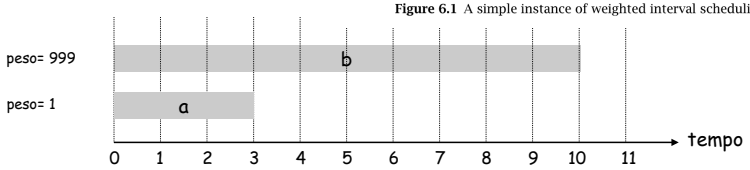


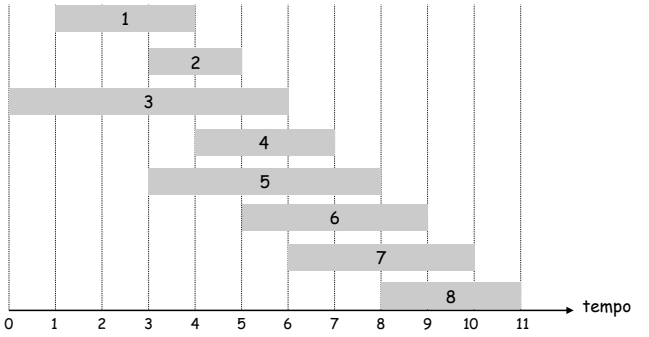
Figure 6.1 A simple instance of weighted interval scheduling.



Schedulazione degli intervalli pesati

Notazione. Labelare job per tempo di fine: $f_1 \leq f_2 \leq \ldots \leq f_n$.
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.



Schedulazione degli intervalli pesati

Notazione. Labelare job per tempo di fine: $f_1 \leq f_2 \leq \ldots \leq f_n$.
Definizione. p(j) = il più grande i < j tale che job i è compatibile con j.

Esempio:

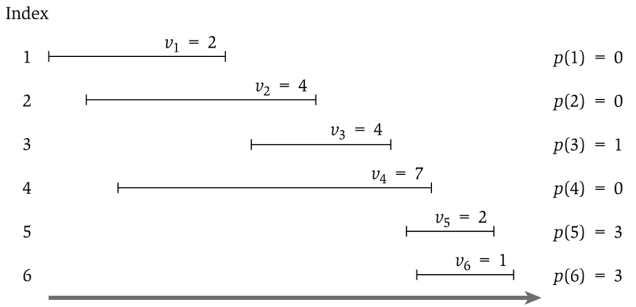


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

Programmazione dinamica: Scelta binaria

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

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

sottostruttura ottima

- Caso 2: OPT non contiene job j.
  - Deve includere soluzione ottimale al problema che consiste dei rimanenti job compatibili 1, 2, ..., j-1

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

Schedulazione degli intervalli pesati: Algoritmo di forza bruta

Algoritmo di forza bruta.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

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

Computare p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(vj + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```


Schedulazione degli intervalli pesati: Algoritmo di forza bruta

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

Esempio.

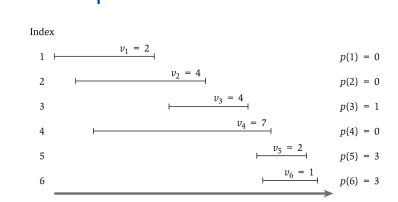


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

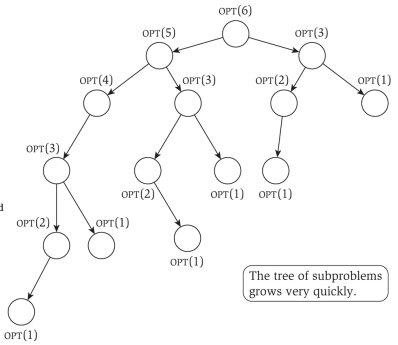
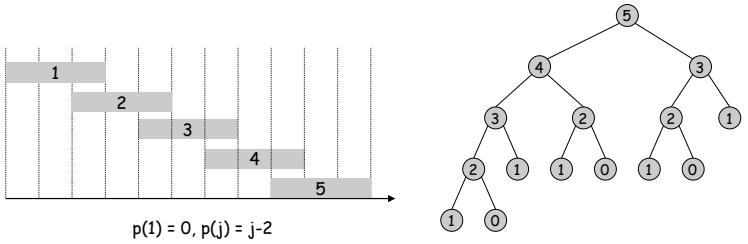


Figure 6.3 The tree of subproblems called by `Compute-Opt` on the problem instance of Figure 6.2.

Schedulazione degli intervalli pesati: Algoritmo di forza bruta

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

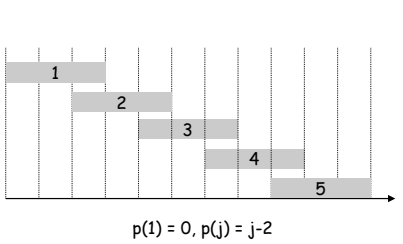
Esempio. Numero di chiamate ricorsive per una famiglia di istanze cresce come la sequenza di Fibonacci.

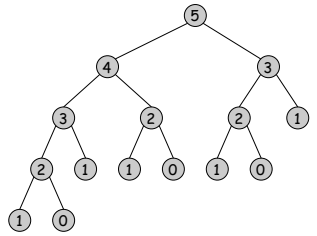


Schedulazione degli intervalli pesati: Algoritmo di forza bruta

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

Esempio. Numero di chiamate ricorsive per una famiglia di istanze cresce come la sequenza dei numeri di Fibonacci.





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

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

$$\varphi = \frac{1+\sqrt{5}}{2} = 1{,}61803\ldots$$

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

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, s1,…,sn , f1,…,fn , v1,…,vn

Ordinare job per tempo di fine f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = 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(vj+M-Compute-Opt(p(j)), M-Compute-Opt(j − 1))
    Return M[j]
  Endif
```


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(·) : O(n) dopo un ordinamento per tempo di inizio.

esercizio da fare a casa

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

- Tempo totale di `M-Compute-Opt(n)` è $O(n)$. ▪

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

Annotazione automatica

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

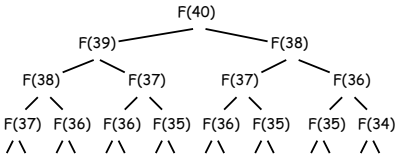
Non succede nei linguaggi imperativi (e.g., Java)?

```
(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)



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.

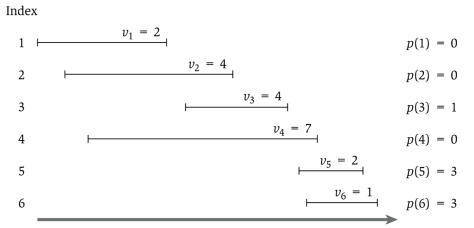


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

$M =$

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

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.

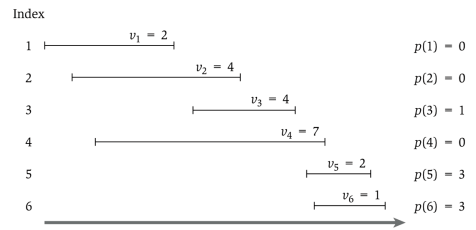


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

E' 6 nella soluzione ottimale?

$M =$

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

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.

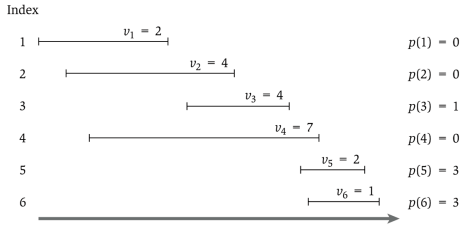


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

E' 6 nella soluzione ottimale? NO

	0	1	2	3	4	5	6
$M =$	0	2	4	6	7	8	8

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

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

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.

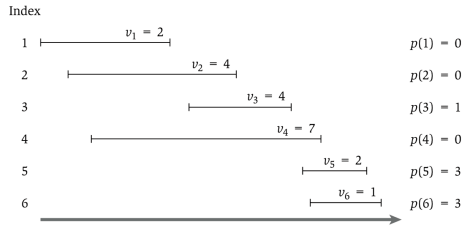


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

E' 5 nella soluzione ottimale?

	0	1	2	3	4	5	6
$M =$	0	2	4	6	7	8	8

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.

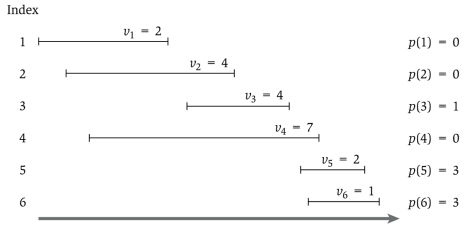


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

E' 5 nella soluzione ottimale? SI

	0	1	2	3	4	5	6
$M =$	0	2	4	6	7	8	8

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

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

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.

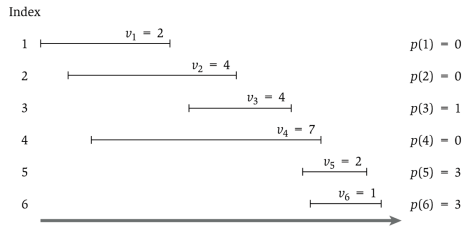


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

E' 3 nella soluzione ottimale?

	0	1	2	3	4	5	6
$M =$	0	2	4	6	7	8	8

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.

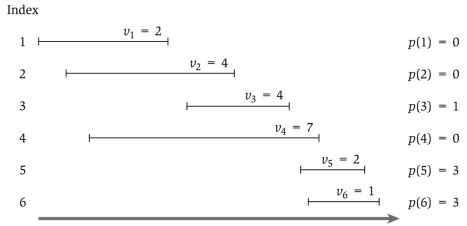


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

E' 3 nella soluzione ottimale? SI

	0	1	2	3	4	5	6
$M =$	0	2	4	6	7	8	8

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

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

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.

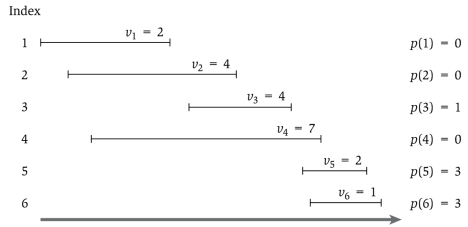


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

E' 1 nella soluzione ottimale? SI

	0	1	2	3	4	5	6
$M =$	0	2	4	6	7	8	8

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

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

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 (v_j + M[p(j)] > M[j-1])
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```

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

Schedulazione degli intervalli pesati: Approccio bottom-Up

Programmazione dinamica bottom-up.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Ordinare job per tempo di fine f_1 ≤ f_2 ≤ ... ≤ f_n.

Computa p(1), p(2), …, p(n)

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

Complessità O(n).

Schedulazione degli intervalli pesati: Approccio bottom-Up

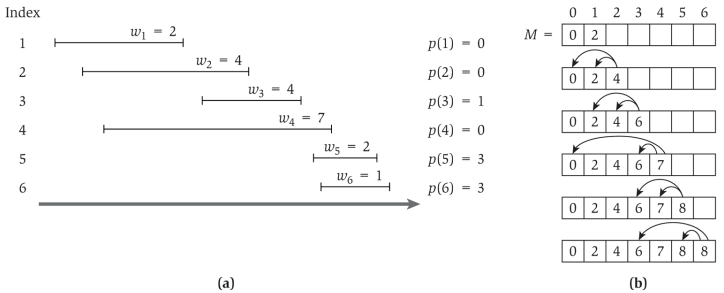


Figure 6.5 Part (b) shows the iterations of `Iterative-Compute-Opt` on the sample instance of Weighted Interval Scheduling depicted in part (a).

Esercizio: calcolo p(j)

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

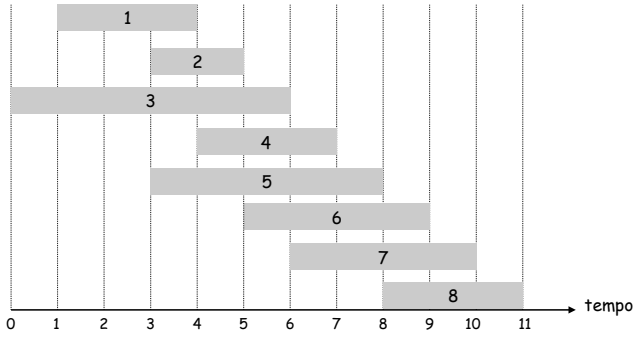
Quindi, dati i job ordinati per tempo di inizio, come computare in tempo lineare i valori p(j)?

```
Input: n, s1,…,sn , f1,…,fn , π

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

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


Esercizio: calcolo p(j), esempio



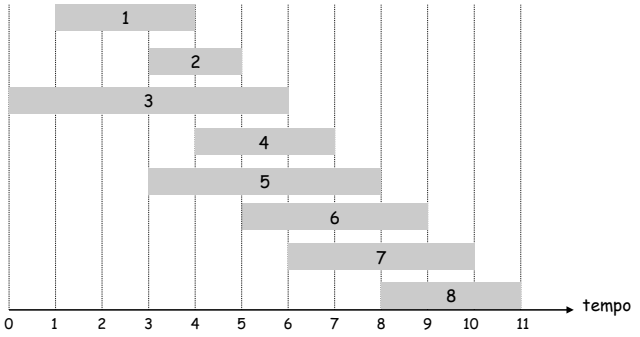
$s_{\pi(i)}$	0	1	3	3	4	5	6	8
π(i)	3	1	2	5	4	6	7	8

Iniziamo con Π(1)
$s_{\pi(1)}$=0 ed f_1=4
Quindi p(Π(1))=0, cioè p(3)=0

f_i	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

Esercizio: calcolo p(j), esempio



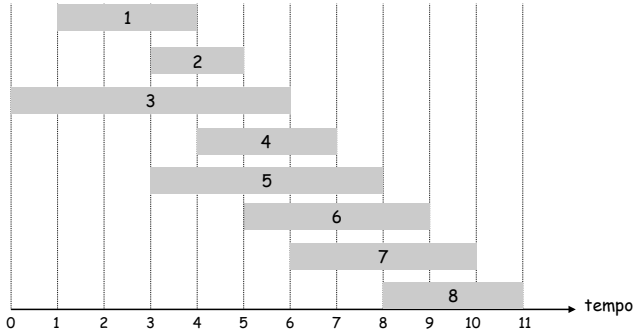
$s_{\pi(i)}$	0	1	3	3	4	5	6	8
π(i)	3	1	2	5	4	6	7	8

Continuiamo con Π(2)
$s_{\pi(2)}$=1 ed f_1=4
Quindi p(Π(2))=0, cioè p(1)=0

f_i	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

Esercizio: calcolo p(j), esempio



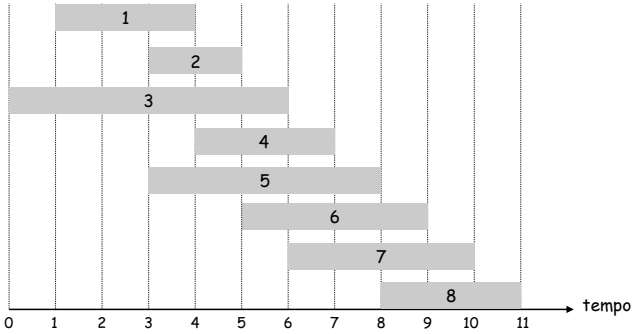
$s_{\pi(i)}$	0	1	3	3	4	5	6	8
$\pi(i)$	3	1	2	5	4	6	7	8

Continuiamo con $\Pi(3)$
$s_{\pi(3)}=3$ ed $f_1=4$
Quindi $p(\Pi(3))=0$, cioè $p(2)=0$

f_i	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

Esercizio: calcolo p(j), esempio



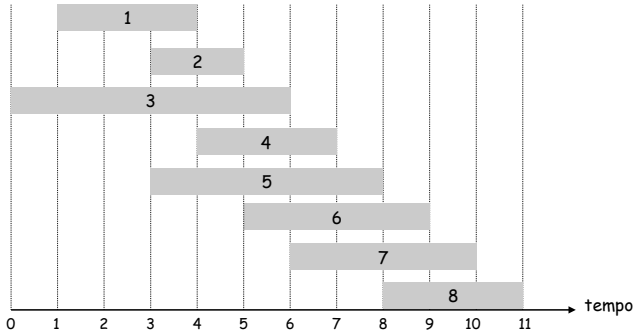
$s_{\pi(i)}$	0	1	3	3	4	5	6	8
$\pi(i)$	3	1	2	5	4	6	7	8

Continuiamo con $\Pi(4)$
$s_{\pi(4)}=3$ ed $f_1=4$
Quindi $p(\Pi(4))=0$, cioè $p(5)=0$

f_i	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

Esercizio: calcolo p(j), esempio



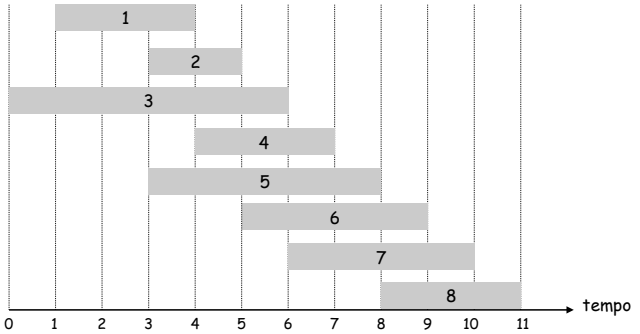
$s_{\pi(i)}$	0	1	3	3	4	5	6	8
$\pi(i)$	3	1	2	5	4	6	7	8

Continuiamo con $\Pi(5)$
$S_{\pi(5)}=4$ ed $f_1=4$, $f_2=5$
Quindi $p(\Pi(5))=1$, cioè $p(4)=1$

f_i	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

Esercizio: calcolo p(j), esempio



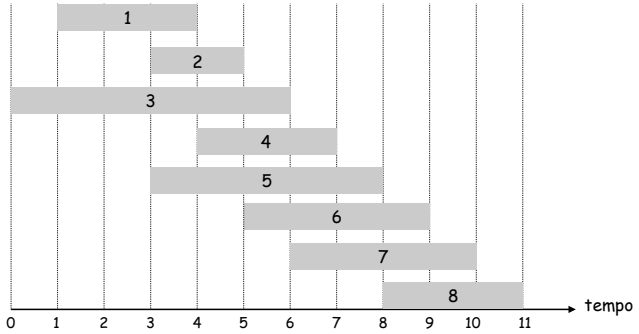
$s_{\pi(i)}$	0	1	3	3	4	5	6	8
$\pi(i)$	3	1	2	5	4	6	7	8

Continuiamo con $\Pi(6)$
$S_{\pi(6)}=5$ ed $f_1=4$, $f_2=5$, $f_3=6$
Quindi $p(\Pi(6))=1$, cioè $p(6)=2$

f_i	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

Esercizio: calcolo p(j), esempio



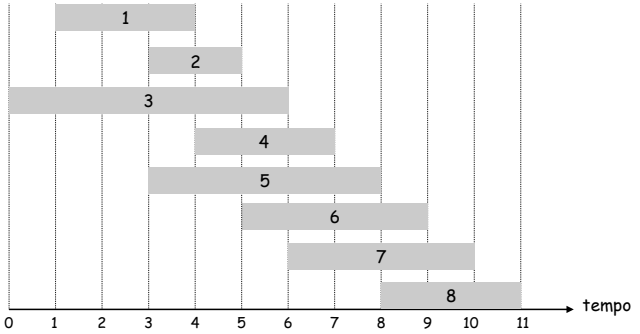
$s_{\pi(i)}$	0	1	3	3	4	5	6	8
$\pi(i)$	3	1	2	5	4	6	7	8

Continuiamo con $\Pi(7)$
$S_{\pi(7)}$=6 ed f_2=5, f_3=6, f_4=7
Quindi p(Π(7))=3, cioè p(7)=3

f_i	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

Esercizio: calcolo p(j), esempio



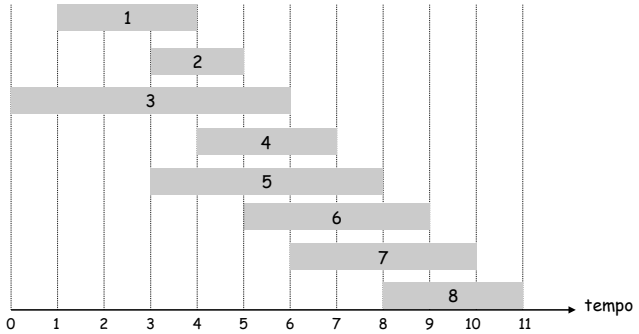
$s_{\pi(i)}$	0	1	3	3	4	5	6	8
$\pi(i)$	3	1	2	5	4	6	7	8

Terminiamo con $\Pi(8)$
$S_{\pi(8)}$=8 ed f_3=6, f_4=7, f_5=8, f_6=9
Quindi p(Π(8))=5, cioè p(8)=5

f_i	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

Esercizio: calcolo p(j), esempio



$s_{\pi(i)}$	0	1	3	3	4	5	6	8
$\pi(i)$	3	1	2	5	4	6	7	8

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

f_i	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

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 , π

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

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

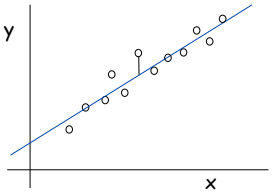
Non può essere più grande del valore j precedente!

6.3 Segmented Least Squares

Least squares.

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

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



Soluzione. Sappiamo che l' errore minimo si ha quando

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

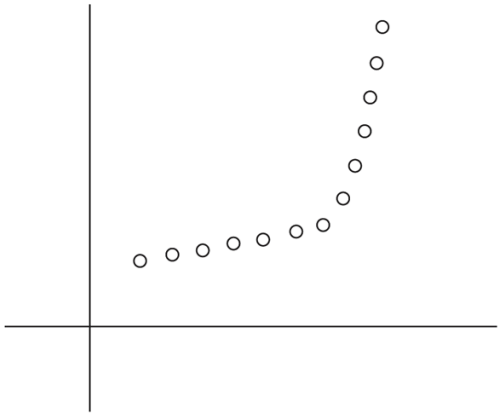


Figure 6.7 A set of points that lie approximately on two lines.

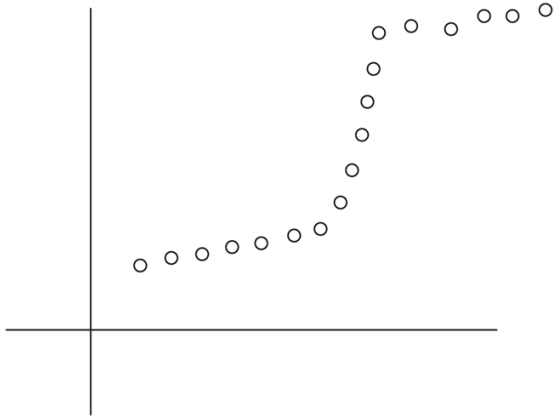


Figure 6.8 A set of points that lie approximately on three lines.

Segmented Least Squares

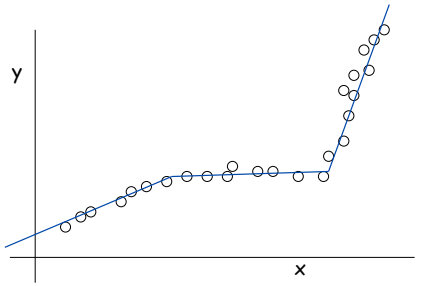
Segmented least squares.

- Punti si trovano approssimativamente su una sequenza di segmenti.
- Dati n punti nel piano (x_1, y_1), (x_2, y_2) , . . . , (x_n, y_n) con $x_1 < x_2 < \ldots < x_n$, 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

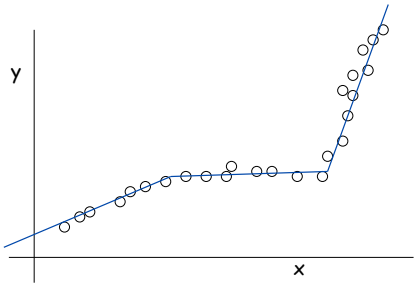
bontà dell' approssimazione



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) , . . . , (x_n, y_n) con $x_1 < x_2 < \ldots < x_n$, 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 + c L, per qualche costante c > 0.



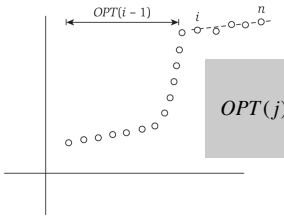
Programmazione dinamica: Multiway Choice

Notazione.

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

Per computare OPT(j):

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



$$OPT(j) = \begin{cases} 0 & \text{se } j = 0 \\ \min_{1 \le i \le j} \{ e(i,j) + c + OPT(i-1) \} & \text{altrimenti} \end{cases}$$

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

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 pi,…, 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

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 pi,…, 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 e(i,j) per j-i=1, j-i=2, ….
e(i,j) da e(i,j-1) in tempo costante

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

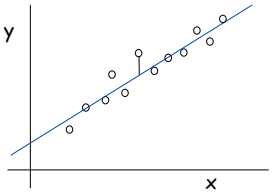
Esercizio: calcolare tutti i valori e(i,j) in $O(n^2)$

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, ….
- calcolare e(i,j) da e(i,j-1) in tempo costante

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

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



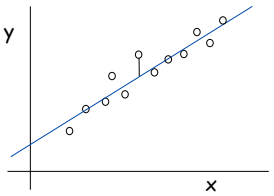
Esercizio: calcolare tutti i valori e(i,j) in $O(n^2)$

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, ….
- calcolare e(i,j) da e(i,j-1) in tempo costante

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

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



Soluzione: calcolare $\sum_{k=i}^{j} x_k, \ \sum_{k=i}^{j} y_k, \ \sum_{k=i}^{j} x_k y_k, \ \sum_{k=i}^{j} x_k^2, \ \sum_{k=i}^{j} y_k^2$

Scrivere l'errore come

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

Segmented Least Squares: Trovare una soluzione

```
Find-Segments(j)
  If j = 0 then
    Output nothing
  Else
    Find an i that minimizes ei,j + C + M[i − 1]
    Output the segment {pi, . . . , pj} and the result of
          Find-Segments(i − 1)
  Endif
```

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 ⇒ algoritmo greedy non ottimale.

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 ⇒ algoritmo greedy non ottimale.

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 ⇒ algoritmo greedy non ottimale.

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!

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

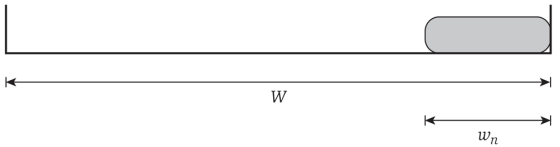


Figure 6.10 After item n is included in the solution, a weight of w_n is used up and there is $W - w_n$ available weight left.

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

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

Problema dello zaino: approccio bottom-up

Utilizzare un array n-per-W.

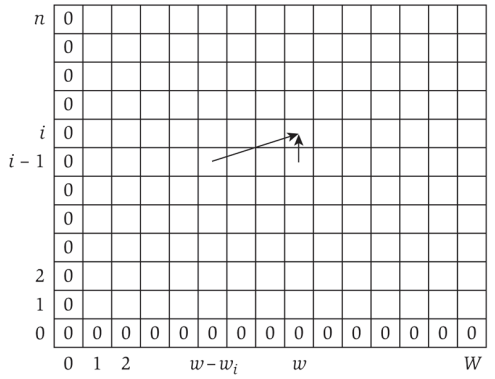


Figure 6.11 The two-dimensional table of OPT values. The leftmost column and bottom row is always 0. The entry for $\text{OPT}(i, w)$ is computed from the two other entries $\text{OPT}(i-1, w)$ and $\text{OPT}(i-1, w-w_i)$, as indicated by the arrows.

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]
```


Problema dello zaino: esempio algoritmo

Knapsack size $W = 6$, items $w_1 = 2$, $w_2 = 2$, $w_3 = 3$

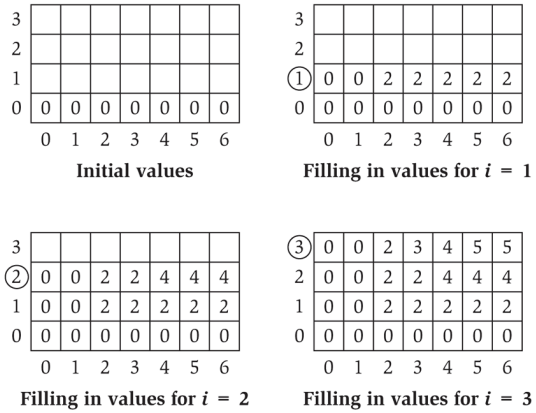


Figure 6.12 The iterations of the algorithm on a sample instance of the Subset Sum Problem.

Problema dello zaino: esempio algoritmo

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	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

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

W = 11

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

Problema dello zaino: Complessità

Complessità. Θ(n W).

- 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!

Problema dello zaino: Esercizio 1

Si descriva ed analizzi un algoritmo per la seguente variazione del problema dello zaino: Dati n oggetti di peso $w_1,w_2,...,w_n$ e valore $v_1,v_2,...,v_n$ 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.)

Problema dello zaino: Esercizio 2

Si descriva ed analizzi un algoritmo per la seguente variazione del problema dello zaino: Dati n oggetti di peso $w_1,w_2,...,w_n$ e valore $v_1,v_2,...,v_n$ 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.)

Problema dello zaino: Esercizio 3

Si descriva ed analizzi un algoritmo per la seguente variazione del problema dello zaino: Dati n oggetti di peso $w_1,w_2,...,w_n$ e valore $v_1,v_2,...,v_n$ 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,...,n-1).

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

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

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)

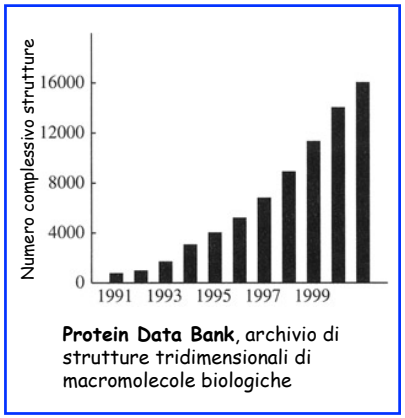
Corsa ai brevetti per Celera ed alla pubblicazione degli accademici per evitare i brevetti

Corsa finita alla pari

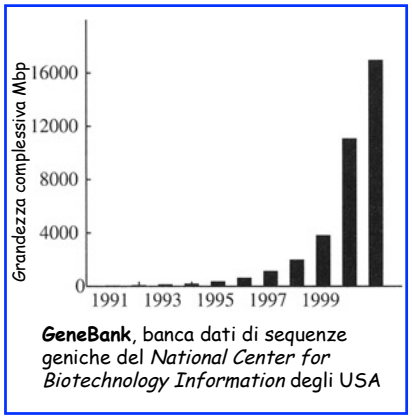
26 giugno 2000, Tony Blair e Bill Clinton annunciano il sequenziamento completo del genoma umano

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

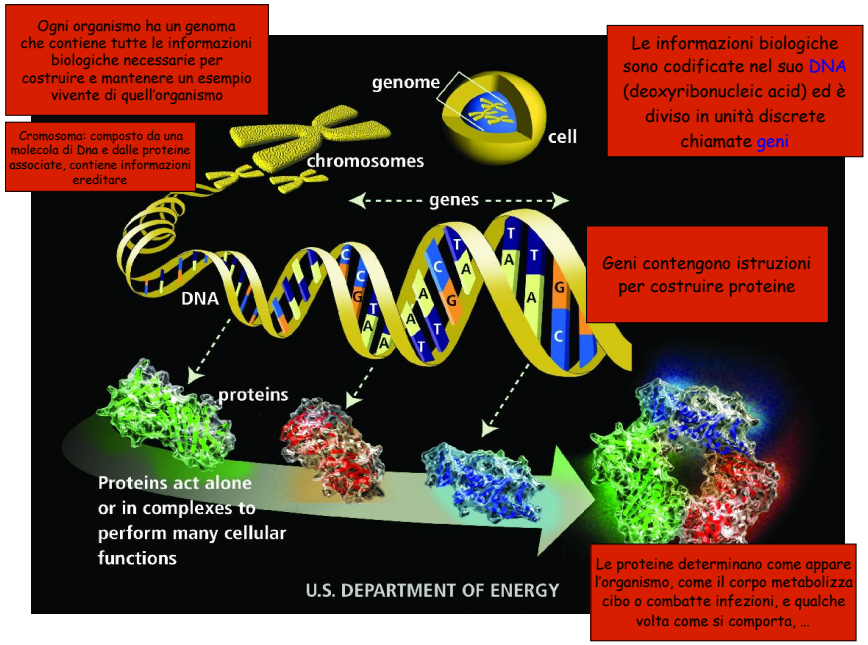
Crescita Banche Dati



Protein Data Bank, archivio di strutture tridimensionali di macromolecole biologiche



GeneBank, banca dati di sequenze geniche del *National Center for Biotechnology Information* degli USA



Genoma

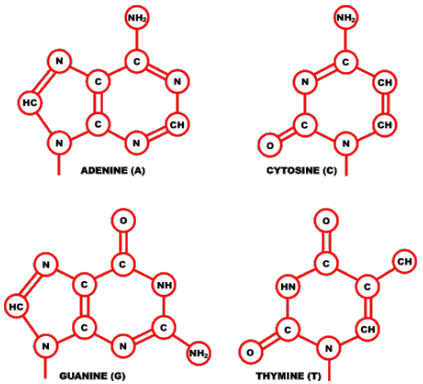
Ogni organismo ha un genoma che contiene tutte le informazioni biologiche necessarie per costruire mantenere un esempio vivente di quell'organismo

Le informazioni biologiche sono codificate nel suo DNA (deoxyribonucleic acid) ed è diviso in unità discrete chiamate geni

Geni contengono istruzioni per costruire proteine richieste dall'organismo

Le proteine determinano, come appare l'organismo, come il corpo metabolizza cibo o combatte infezioni, e qualche volta come si comporta, …

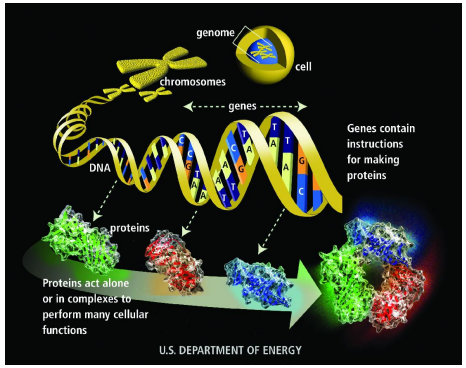
Le 4 basi del DNA

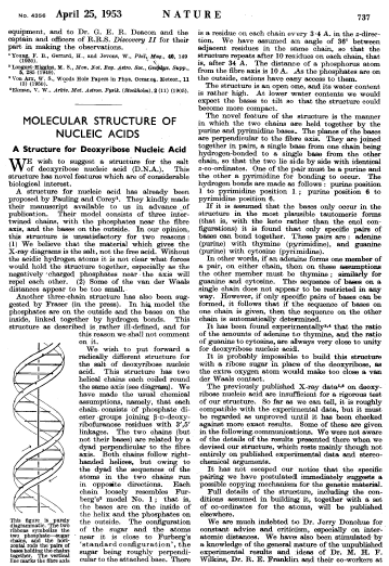


DNA

Molecola formata da 2 catene a forma di doppia elica

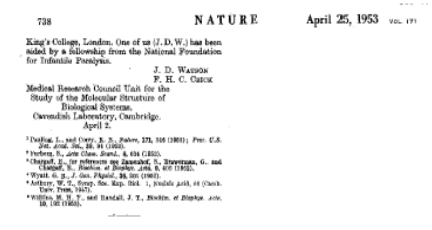
Catene connesse da basi complementari A-T e C-G





MOLECULAR STRUCTURE OF NUCLEIC ACIDS

A Structure for Deoxyribose Nucleic Acid



James Watson e Francis Crick

Ribonucleic acid (RNA)

Simile al DNA.

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

RNA. Stringa $B = b_1 b_2 \ldots b_n$ su alfabeto { A, C, G, U }.

RNA Secondary Structure

RNA. Stringa $B = b_1 b_2 \ldots b_n$ 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

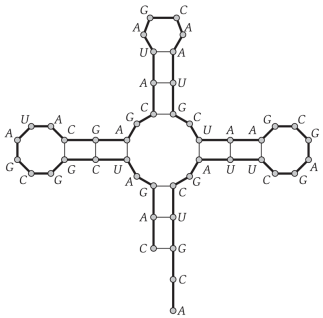


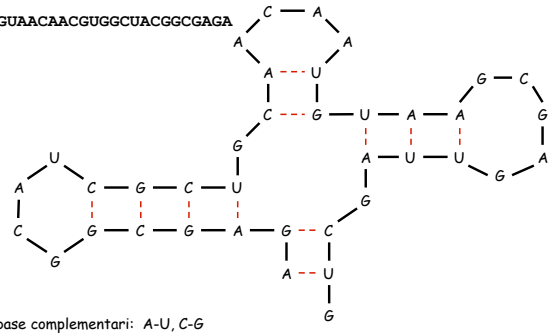
Figure 6.13 An RNA secondary structure. Thick lines connect adjacent elements of the sequence; thin lines indicate pairs of elements that are matched.

RNA Secondary Structure

RNA. Stringa B = $b_1 b_2 \ldots b_n$ 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: GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA

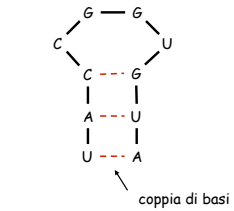


coppie di base complementari: A-U, C-G

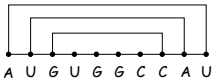
RNA Secondary Structure

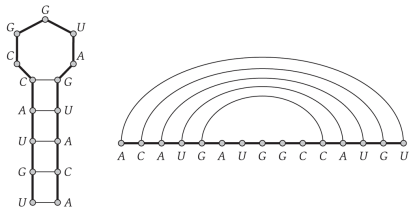
Struttura secondaria. Un insieme di coppie S = { (b_i, b_j) } che soddisfa:

- [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.



coppia di basi





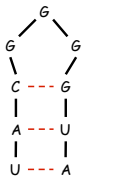
(a) (b)

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.

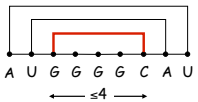
RNA Secondary Structure

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

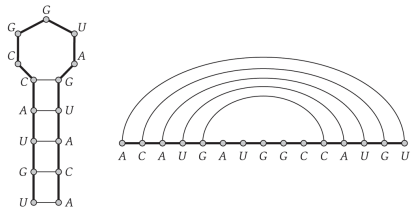
- [Watson-Crick.] S è un matching ed ogni coppia in S è un complemento Watson-Crick: A-U, U-A, C-G, oppure G-C.
- [No sharp turns.] Gli elementi di ogni coppia sono separati da almeno 4 basi. Se $(b_i, b_j) \in S$, allora $i < j - 4$.



sharp turn



≤4



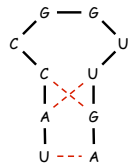
(a) (b)

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.

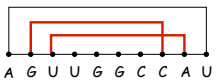
RNA Secondary Structure

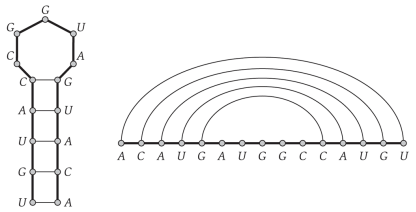
Struttura secondaria. Un insieme di coppie S = { (b_i, b_j) } che soddisfa:

- [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 $i < k < j < l$.



crossing





(a) (b)

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.

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 $i < k < j < l$.

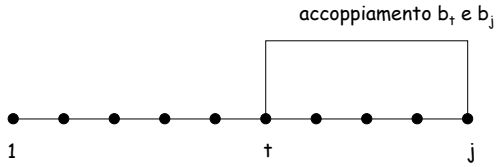
Energia libera. L'ipotesi usuale è che una molecola RNA formerà la struttura secondaria con l'energia libera totale ottimale.

approssimativamente il numero di coppie di basi

Obiettivo. Data una molecola RNA $B = b_1 b_2 \ldots b_n$, trovare una struttura secondaria S che massimizza il numero di coppie di basi.

RNA Secondary Structure: Sottoproblemi

Primo tentativo. OPT(j) = massimo numero di coppie di basi in una struttura secondaria della sottostringa $b_1 b_2 \ldots b_j$.



Difficoltà. Abbiamo due sottoproblemi:

- Trovare la struttura secondaria in: $b_1 b_2 \ldots b_{t-1}$. ← OPT(t-1)
- Trovare la struttura secondaria in: $b_{t+1} b_{t+2} \ldots b_{j-1}$. ← abbiamo bisogno di più sottoproblemi

Programmazione dinamica su intervalli

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

- 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
  - $OPT(i, j) = 1 + \max_{i \leq t < j-4} \{ OPT(i, t-1) + OPT(t+1, j-1) \}$

b_t e b_j sono complementi Watson-Crick, cioè A-U, U-A, C-G, oppure G-C.

Programmazione dinamica su intervalli

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

- 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.

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

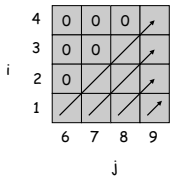
Bottom Up Dynamic Programming Over Intervals

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]
}
```

usare ricorrenza

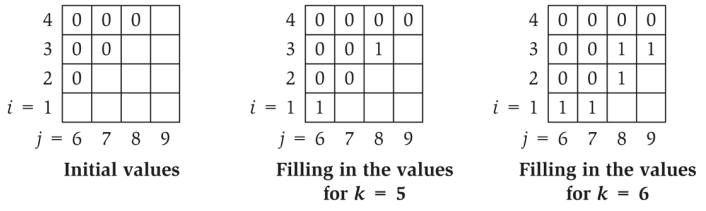


Complessità. $O(n^3)$.

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

RNA sequence *ACCGGUAGU*

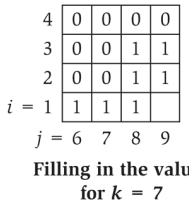
A C C G G U A G U



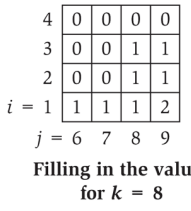
Initial values

Filling in the values for $k = 5$

Filling in the values for $k = 6$



Filling in the values for $k = 7$

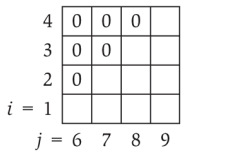


Filling in the values for $k = 8$

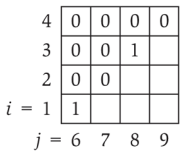
Figure 6.16 The iterations of the algorithm on a sample instance of the RNA Secondary Structure Prediction Problem.

RNA sequence *ACCGGUAGU*

A C C G G U A G U



Initial values



Filling in the values for $k = 5$

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

$$OPT(4,9) = \max \begin{cases} OPT(4, 8) = 0 \\ \max_{4 \le t < 5} \{ 1 + OPT(4, t-1) + OPT(t+1, 8) \} = 0 \\ \quad b_t b_j \text{ complementi} \end{cases}$$

G G U A G U

$$OPT(3,8) = \max \begin{cases} OPT(3, 7) = 0 \\ \max_{3 \le t < 4} \{ 1 + OPT(1, t-1) + OPT(t+1, 7) \} = 1 \\ \quad b_t b_j \text{ complementi} \end{cases}$$

C G G U A G

$$OPT(2,7) = \max \begin{cases} OPT(2, 6) = 0 \\ \max_{2 \le t < 3} \{ 1 + OPT(2, t-1) + OPT(t+1, 6) \} = 0 \\ \quad b_t b_j \text{ complementi} \end{cases}$$

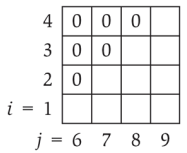
C C G G U A

$$OPT(1,6) = \max \begin{cases} OPT(1, 5) = 0 \\ \max_{1 \le t < 2} \{ 1 + OPT(1, t-1) + OPT(t+1, 5) \} = 1 \\ \quad b_t b_j \text{ complementi} \end{cases}$$

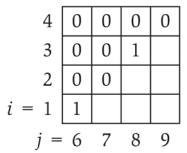
A C C G G U

RNA sequence *ACCGGUAGU*

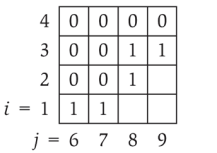
A C C G G U A G U



Initial values



Filling in the values for $k = 5$



Filling in the values for $k = 6$

$$OPT(3,9) = \max \begin{cases} OPT(3, 8) = 1 \\ \max_{3 \le t < 5} \{ 1 + OPT(3, t-1) + OPT(t+1, 8) \} = 0 \\ \quad b_t b_j \text{ complementi} \end{cases}$$

C G G U A G U

$$OPT(2,8) = \max \begin{cases} OPT(2, 7) = 1 \\ \max_{2 \le t < 4} \{ 1 + OPT(2, t-1) + OPT(t+1, 7) \} = 1 \\ \quad b_t b_j \text{ complementi} \end{cases}$$

C C G G U A G oppure C C G G U A G

$$OPT(2,8) = \max \begin{cases} OPT(2, 7) = 1 \\ \max_{2 \le t < 4} \{ 1 + OPT(2, t-1) + OPT(t+1, 7) \} = 1 \\ \quad b_t b_j \text{ complementi} \end{cases}$$

A C C G G U A

Figure 6.16 The iterations of the algorithm on a sample instance of the RNA Secondary Structure Prediction Problem.

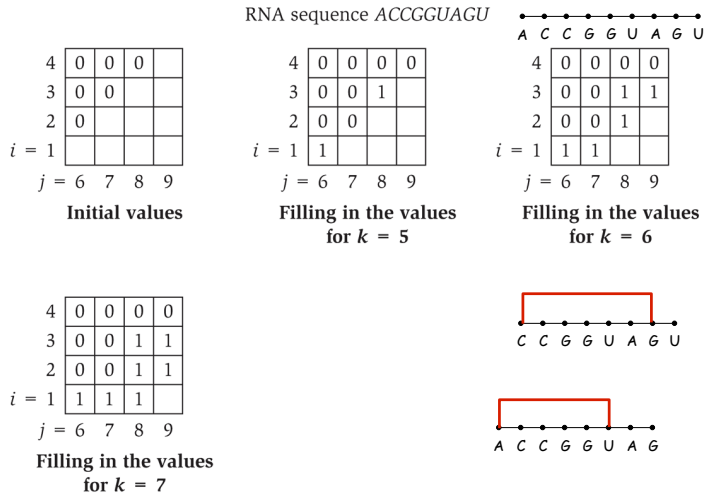


Figure 6.16 The iterations of the algorithm on a sample instance of the RNA Secondary Structure Prediction Problem.

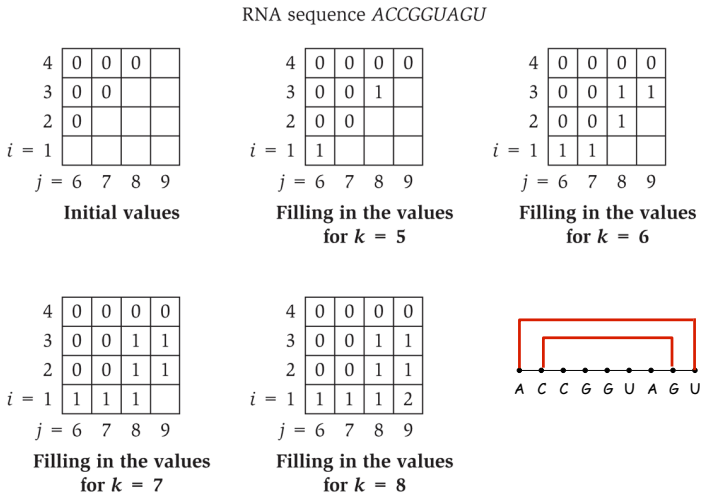


Figure 6.16 The iterations of the algorithm on a sample instance of the RNA Secondary Structure Prediction Problem.

Trovare una soluzione

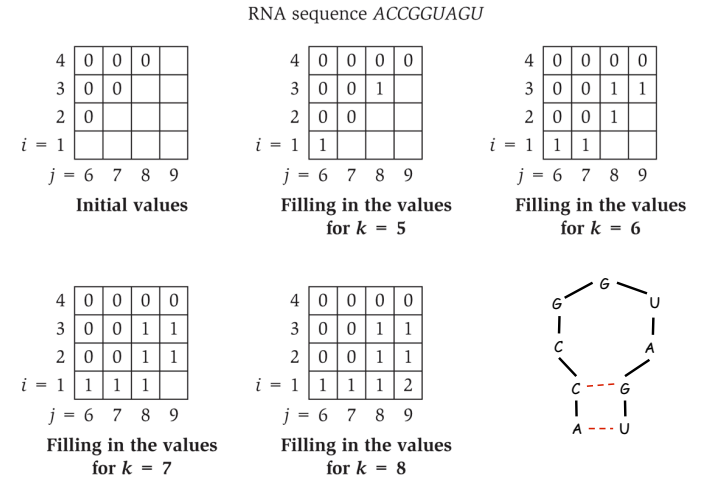


Figure 6.16 The iterations of the algorithm on a sample instance of the RNA Secondary Structure Prediction Problem.

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.

6.6 Sequence Alignment

Similarità di stringhe

Quanto sono simili due stringhe?

- **ocurrance**
- **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

Edit Distance

Applicazioni.

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

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

- Penalità per gap δ; penalità per mismatch α_{pq}.
- Costo = somma delle penalità di gap e mismatch.

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

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

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

$2\delta + \alpha_{CA}$

Sequence Alignment

Obiettivo: Date due stringhe $X = x_1 x_2 \ldots x_m$ e $Y = y_1 y_2 \ldots y_n$ trovare un allineamento di costo minimo.

Definizione. Un allineamento M è un insieme di coppie ordinate x_i-y_j tali che ogni elemento occorre in al massimo una coppia e senza *cross*.

Definizione. La coppia x_i-y_j e $x_{i'}$-$y_{j'}$ cross se $i < i'$, ma $j > j'$.

$$\text{costo}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i : x_i \text{ non accoppiato}} \delta + \sum_{j : y_j \text{ non accoppiato}} \delta}_{\text{gap}}$$

Esempio: CTACCG vs. TACATG.

Soluzione: M = x_2-y_1, x_3-y_2, x_4-y_3, x_5-y_4, x_6-y_6.

	x_1	x_2	x_3	x_4	x_5		x_6
	C	T	A	C	C	-	G
	-	T	A	C	A	T	G
	y_1	y_2	y_3	y_4	y_5		y_6

Sequence Alignment: Struttura del problema

Definizione. OPT(i, j) = min costo allineamento $x_1 x_2 \ldots x_i$ e $y_1 y_2 \ldots y_j$.

- Caso 1: OPT accoppia x_i-y_j.
  - mismatch per x_i-y_j + min costo allineamento $x_1 x_2 \ldots x_{i-1}$ e $y_1 y_2 \ldots y_{j-1}$
- Caso 2a: OPT lascia x_i senza accoppiamento.
  - gap per x_i e minimo costo allineamento $x_1 x_2 \ldots x_{i-1}$ e $y_1 y_2 \ldots y_j$
- Caso 2b: OPT lascia y_j senza accoppiamento.
  - gap per y_j e minimo costo allineamento $x_1 x_2 \ldots x_i$ e $y_1 y_2 \ldots y_{j-1}$

$$OPT(i, j) = \begin{cases} j\delta & \text{se } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{altrimenti} \\ i\delta & \text{se } j = 0 \end{cases}$$

Sequence Alignment: 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]
}
```


Sequence Alignment: Algoritmo

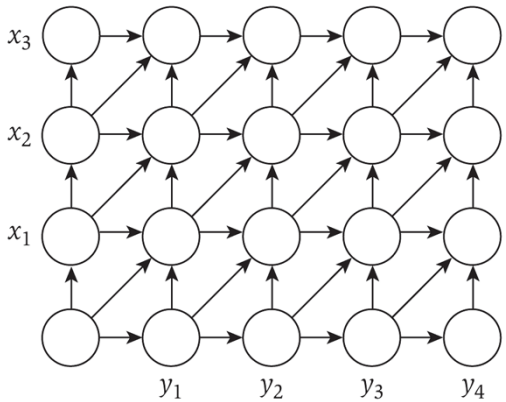


Figure 6.17 A graph-based picture of sequence alignment.

Sequence Alignment: Esempio algoritmo

Penalità per gap $\delta = 2$

Penalità per mismatch

- $\alpha_{vocale,vocale} = 1$
- $\alpha_{consonante,consonante} = 1$
- $\alpha_{vocale,consonante} = 3$

M[4,4] = min(3+M[3,3], 2+M[3,4], 2+M[4,3])
= min(8,7,6)
= 6

m	e	a	n	-
n	-	a	m	e
1	2		1	2

= 6

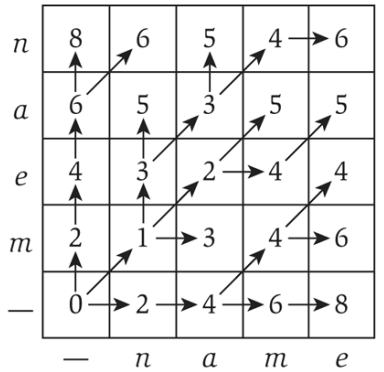


Figure 6.18 The OPT values for the problem of aligning the words *mean* to *name*.

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

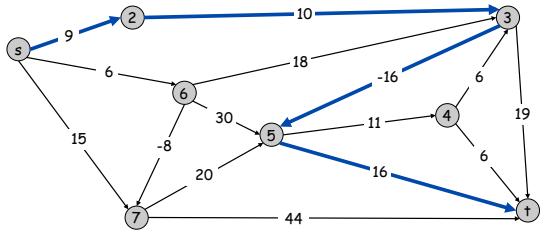
Problema del cammino più corto. Dato un grafo diretto $G = (V, E)$, con peso degli archi c_{vw}, trovare il cammino più corto dal nodo s al nodo t.

anche pesi negativi

Esempio. I nodi rappresentano agenti in ambito finanziario

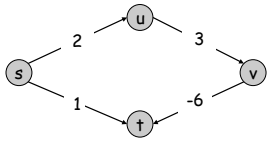
c_{vw} è il costo di una transazione nella quale si acquista da v e si vende a w.

Un cammino rappresenta una successione di transazioni.



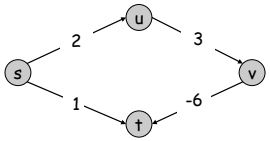
Shortest Paths: Approcci che non funzionano

Algoritmo di Dijkstra. Potrebbe non funzionare con costi negativi.

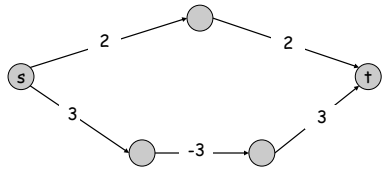


Shortest Paths: Approcci che non funzionano

Algoritmo di Dijkstra. Potrebbe non funzionare con costi negativi.

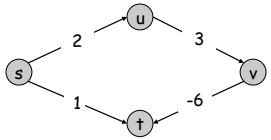


Scalare il peso degli archi. Aggiungere una costante ad ogni peso degli archi potrebbe non funzionare.

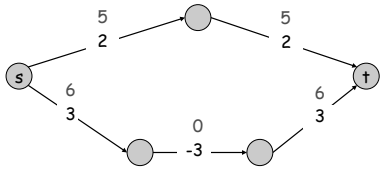


Shortest Paths: Approcci che non funzionano

Algoritmo di Dijkstra. Potrebbe non funzionare con costi negativi.

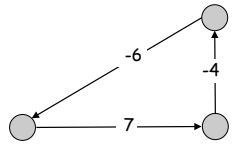


Scalare il peso degli archi. Aggiungere una costante ad ogni peso degli archi potrebbe non funzionare.

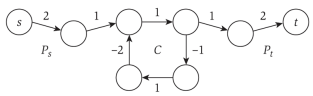


Shortest Paths: Ciclo con costo negativo

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.



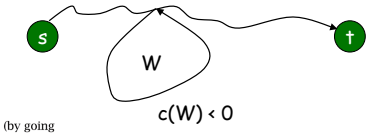


Figure 6.20 In this graph, one can find *s-t* paths of arbitrarily negative cost (by going around the cycle *C* many times).

Assumiamo che il grafo non contenga cicli con costo negativo!

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) = c_{vw} + OPT(i-1, w)

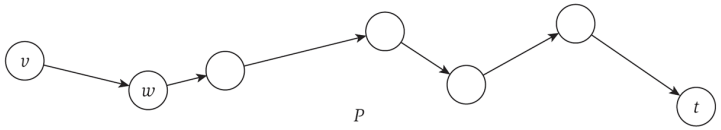


Figure 6.22 The minimum-cost path P from v to t using at most i edges.

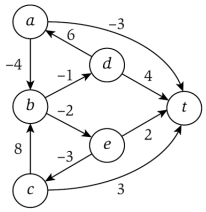
Shortest Paths: Programmazione dinamica

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

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

Nota. Se non ci sono cicli negativi, allora OPT(n-1, v) = lunghezza del cammino v-t più corto.



(a)

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	−3	−3	−4	−6	−6
b	∞	∞	0	−2	−2	−2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

(b)

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

Shortest Paths: Esempio

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

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))

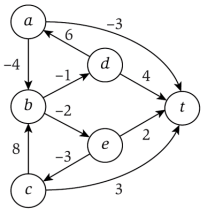
OPT(4,a) = min(OPT(3,a), min(OPT(3,t)+c_{at},OPT(3,b)+c_{ab}) a-b
= min(-4, min(0-3,-2-4))

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

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

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

a - b - e - c - t
-4 -2 -3 +3 = -6



(a)

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	−3	−3	−4	−6	−6
b	∞	∞	0	−2	−2	−2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

(b)

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}) a-b
= min(-4, min(0-3,-2-4))

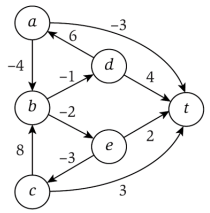
OPT(3,b) = min(OPT(2,b), min(OPT(2,e)+c_{be},OPT(2,d)+c_{bd}) b-e
= min(0, min(0-2,3-1))

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

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

a - b - e - c - t
-4 -2 -3 +3 = -6

Shortest Paths: Esempio



(a)

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	−3	−3	−4	−6	−6
b	∞	∞	0	−2	−2	−2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

(b)

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

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

Mantenere un "successore" per ogni valore della tabella.

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

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

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

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

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

a - b - e - c - t
-4 -2 -3 +3 = -6

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] + c_vw }
}
```


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] + c_vw }
}
```

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

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)

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

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

```
Shortest-Path(G, t) {
   array M[0…n-1,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] + c_vw }
}
```

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 ≤ i archi.

Impatto totale.

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

Riepilogo Cap. 6, Dynamic Programming