Programmazione dinamica

- Progettazione di Algoritmi a.a. 2015-16
- Matricole congrue a 1
- Docente: Annalisa De Bonis

Paradigmi della Progettazione degli Algoritmi

- Greedy. Costruisci una soluzione in modo incrementale, ottimizzando (in modo miope) un certo criterio locale.

- Divide-and-conquer. Suddividi il problema in sottoproblemi, risolvi ciascun sototproblema indipendentemente e combina le soluzioni dei sottoproblemi per formare la soluzione del problema di partenza.

- Programmazione dinamica. Suddividi il problema in un insieme di sottoproblemi che si sovrappongono, cioè che hanno dei sottoproblemi in comune. Costruisci le soluzioni a sottoproblemi via via sempre più grandi in modo da computare la soluzione di un dato sottoproblema un'unica volta.
  - Nel divide and conquer, se due sottoproblemi condividono uno stesso sottoproblema quest'ultimo viene risolto più volte.

Storia della programmazione dinamica

- Bellman. Negli anni '50 è stato il pionere nello studio sistematico della programmazione dinamica.

- Etimologia.
  - Programmazione dinamica = pianificazione nel tempo.

Applicazioni della programmazione dinamica

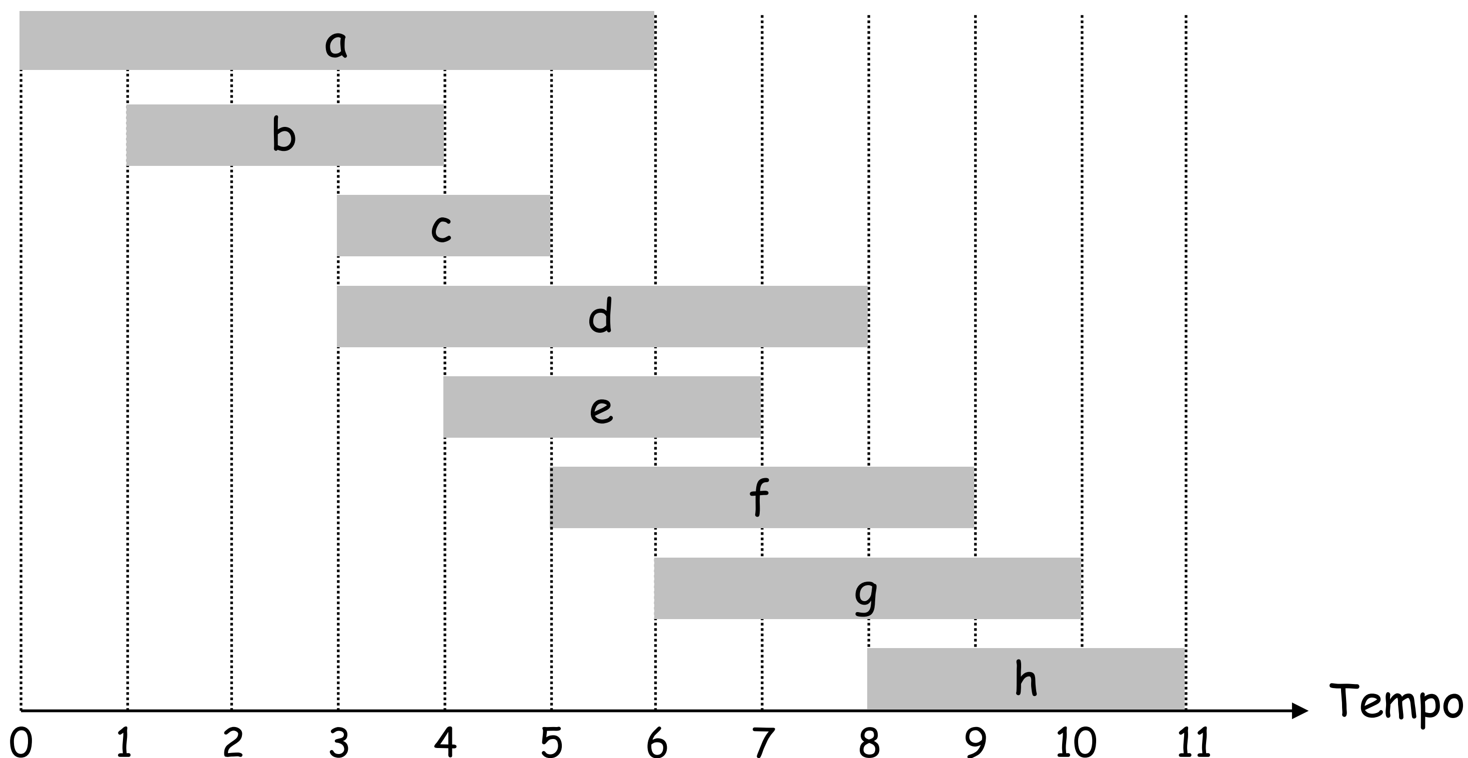
- Aree.
  - Bioinformatica.
  - Teoria dell'informazione
  - Ricerca operativa
  - Informatica teorica
  - Computer graphics
  - Sistemi di Intelligenza Artificiale

- .

Interval Scheduling Pesato

Interval scheduling con pesi

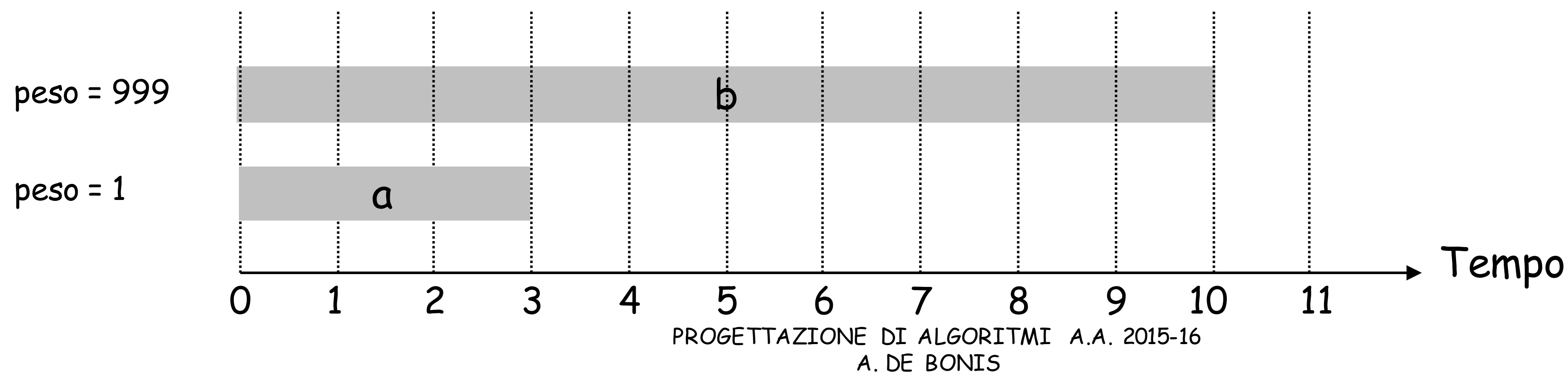
- Job j: comincia al tempo s_j, finisce al tempo f_j, ha associato un valore (peso) v_j .
- Due job sono compatibili se non si sovrappongono
- Obiettivo: trovare il sottoinsieme di job compatibili con il massimo peso totale .



Interval scheduling senza pesi

- L'algoritmo greedy Earliest Finish Time funziona quando tutti i pesi sono uguali ad 1.
  - Considera i job in ordine non decrescente dei tempi di fine
  - Seleziona un job se è compatibile con quelli già selezionati

- Osservazione. L'algoritmo greedy Earliest Finish Time può fallire se i pesi dei job sono valori arbitrari.



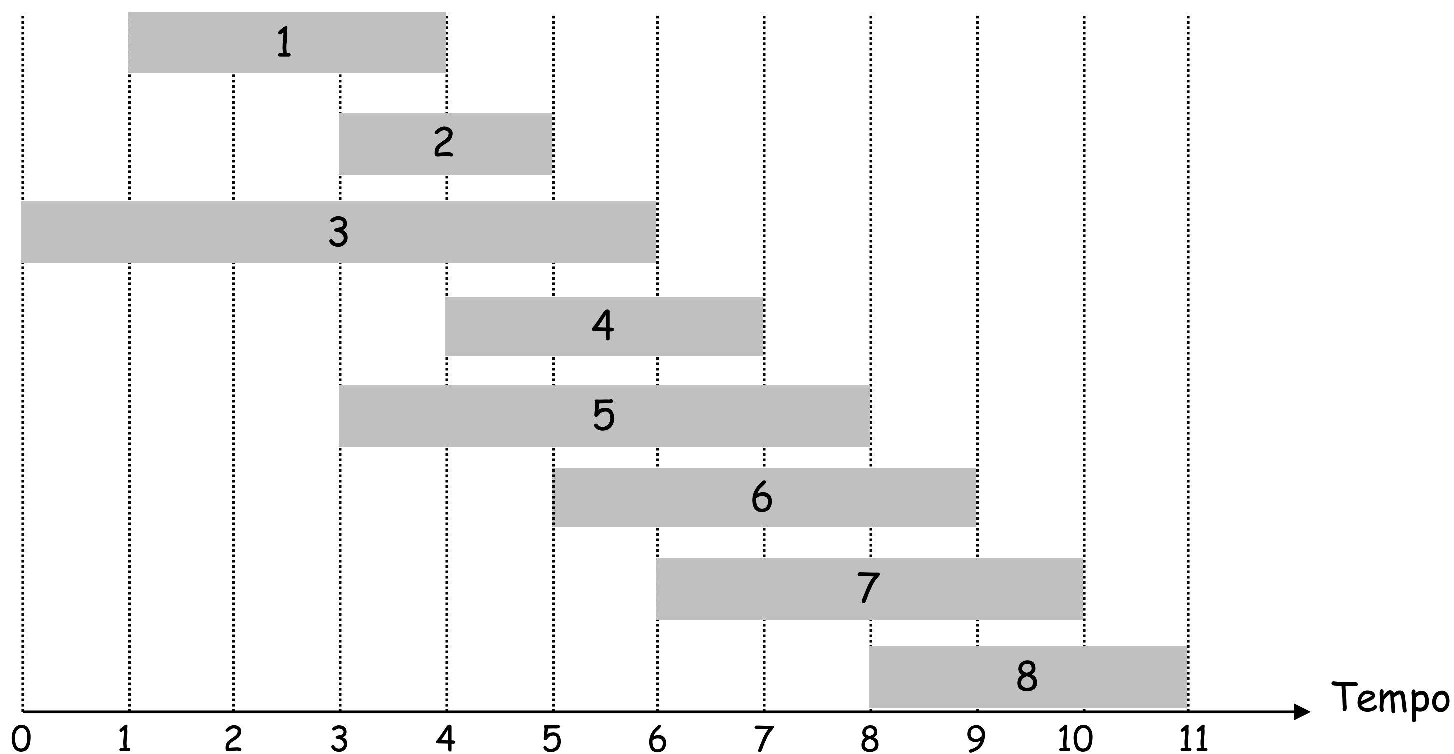
Interval Scheduling Pesato

Notazione. Etichettiamo i job in base al tempo di fine :

$$f_1 \leq f_2 \leq \ldots \leq f_n .$$

Def. p(j) = il più grande indice i < j tale che i è compatibile con j

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



Interval Scheduling Pesato: soluzione basata sulla PD

- Notazione. OPT(j) = valore della soluzione ottima OPT per il problema che consiste nello schedulare le j richieste con i j tempi di fine più piccoli

- Caso 1: OPT seleziona il job j.
  - In questo caso la soluzione non può usare i job incompatibili { p(j) + 1, p(j) + 2, ..., j - 1 }
  - Deve includere la soluzione ottima al problema che consiste nello schedulare i job compatibili 1, 2, ..., p(j)

- Case 2: OPT non seleziona il job j.
  - In questo caso la soluzione deve includere la soluzione ottima al problema che consiste nello schedulare i job 1, 2, ..., j-1

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

Interval Scheduling Pesato: algoritmo ricorsivo inefficiente

- Inizialmente Compute-Opt viene invocato con j=n

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

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute 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))
}
```

Interval Scheduling Pesato: algoritmo ricorsivo inefficiente

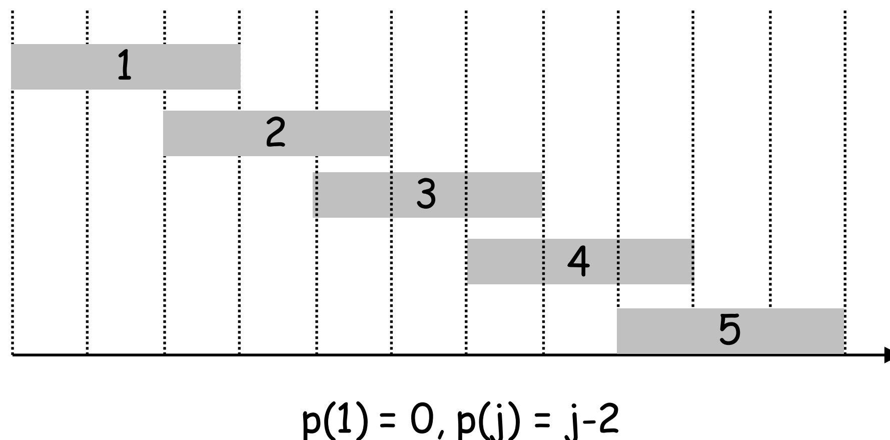
- L'algoritmo computa correttamente OPT(j)
- Dim per induzione.
- Caso base j=0. Il valore restituito è correttamente 0.
- Passo Induttivo. Consideriamo un certo j>0 e supponiamo (ipotesi induttiva) che l'algoritmo produca il valore corretto di OPT(i) per ogni i<j.
- Il valore computato per j dall'algoritmo è

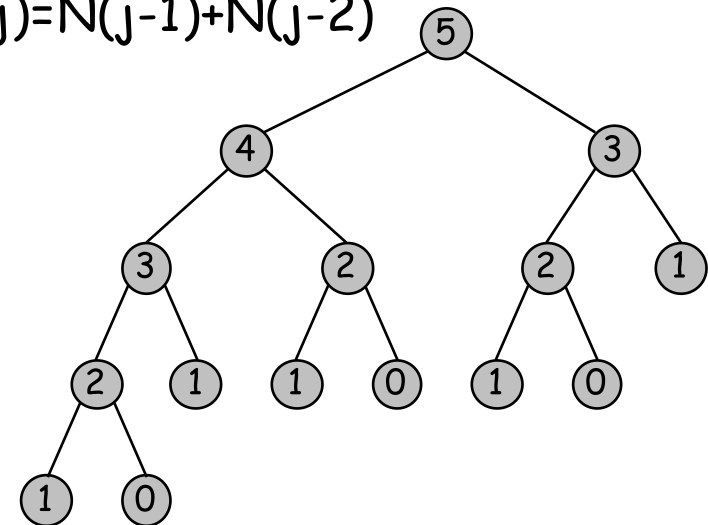
  Compute-Opt(j)= max(v_j + Compute-Opt(p(j)),Compute-Opt(j-1))

- Siccome per ipotesi induttiva **Compute-Opt(p(j))= OPT(p(j))** e
- **Compute-Opt(j-1)= OPT(j-1)** allora si ha
-
- **Compute-Opt(j)= max(v_j + OPT(p(j)), OPT(j-1))= OPT(j)**

Interval Scheduling Pesato: algoritmo ricorsivo inefficiente

- Osservazione. L'algoritmo ricorsivo corrisponde ad un algoritmo di forza bruta perchè ha tempo esponenziale
  - Ciò è dovuto al fatto che
    - ✐ Un gran numero di sottoproblemi sono condivisi da più sottoproblemi
    - ✐ L'algoritmo computa più volte la soluzione ad uno stesso sottoproblema.
- Esempio. In questo esempio il numero di chiamate ricorsive cresce come i numeri di Fibonacci.
- N(j)= numero chiamate ricorsive per j. N(j)=N(j-1)+N(j-2)





Interval Scheduling Pesato: Memoization

- Osservazione: l'algoritmo ricorsivo precedente computa la soluzione di n+1 sottoproblemi soltanto OPT(0),…,OPT(n). Il motivo dell'inefficienza dell'algoritmo è dovuto al fatto che computa la soluzione ad uno stesso problema più volte.

- Memoization. Consiste nell'immagazzinare le soluzioni di ciascun sottoproblema in un'area di memoria accessibile globalmente.

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

Sort jobs by finish times so that 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 Return 0
   if (M[j] is empty)
      M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

inizializzazione (righe da Input a M[j] = empty)

Interval Scheduling pesato: Tempo di Esecuzione

Affermazione. La versione "memoized" dell'algoritmo ha tempo di esecuzione O(n log n).

Fase di inizializzazione: O(n logn)

- Ordinamento in base ai tempi di fine: O(n log n).
- Computazione dei valori p(·): O(n) dopo aver ordinato i job (rispetto ai tempi di inizio e di fine). Siano $a_1,\ldots,a_n$ i job ordinati rispetto ai tempi di inizio e $b_1,\ldots,b_n$ i job ordinati rispetto ai tempi di fine. (si noti che il job con l'i-esimo tempo di inizio non corrisponde necessariamente a quello con l'i-esimo tempo di fine)
  - Si confronta il tempo di fine di b_1 con i tempi di inizio di $a_1,a_2,a_3\ldots$, fino a che non si incontra un job a_j con tempo di inizio $\geq f_1$. Si pone $p'(1)=p'(2)=\ldots=p'(j-1)=0$. Si confronta il tempo di fine di b_2 con i tempi di inizio di $a_j,a_{j+1},a_{j+2}\ldots$, fino a che non si incontra un job a_k con tempo di inizio $\geq f_2$. Si pone $p'(j)=p'(j+1)=p'(j+2)=\ldots=p'(k-1)=1$. Si confronta il tempo di fine di b_3 con i tempi di inizio di $a_k,a_{k+1},a_{k+2}\ldots$, fino a che non si incontra un job a_m con tempo di inizio $\geq f_3$. Si pone $p'(k)=p'(k+1)=p'(k+2)=\ldots=p'(m-1)=2$, e cosi` via.

Continua nel slide successiva

Interval Scheduling pesato: Tempo di Esecuzione

Si noti che i valori p'(j) contengono gli indici dei job ordinati in base ai valori dei tempi di inizio. Per ottenere i valori p(j) basta associare a ciascun job il suo indice nell'ordinamento $b_1,\ldots,b_n$.

Il tempo per calcolare i valori p(1),…,p(n) e` O(n) perche` dopo ogni confronto l'algoritmo passa a considerare o il prossimo job nell'ordinamento $b_1,\ldots,b_n$ (nel caso di confronto tra due job compatibili) o il prossimo job nell'ordinamento $a_1,\ldots,a_n$ (nel caso di confronto tra due job incompatibili).

Continua nel slide successiva

Interval Scheduling pesato: Tempo di Esecuzione

Affermazione: `M-Compute-Opt(n)` richiede $O(n)$

Dim.

- **`M-Compute-Opt(j)`**: escludendo il tempo per le chiamate ricorsive, ciascuna invocazione prende tempo $O(1)$ e fa una delle seguenti cose
  - (i) restituisce il valore esistente di `M[j]`
  - (ii) riempie l'entrata `M[j]` facendo due chiamate ricorsive

- Per stimare il tempo di esecuzione di **`M-Compute-Opt(j)`** dobbiamo stimare il numero totale di chiamate ricorsive innescate da **`M-Compute-Opt(j)`**
  - Abbiamo bisogno di una misura di come progredisce l'algoritmo

- Misura di progressione Φ = # numero di entrate non vuote di `M[]`.
  - inizialmente $\Phi = 0$ e durante l'esecuzione si ha sempre $\Phi \leq n$.
  - per far crescere Φ di 1 occorrono al piu` 2 chiamate ricorsive.
  - quindi per far andare Φ da 0 a j, occorrono al piu` 2j chiamate ricorsive per un tempo totale di $O(j)$

- Il tempo di esecuzione di **`M-Compute-Opt(n)`** e` quindi $O(n)$.
- N.B. $O(n)$, una volta ordinati i job in base ai valori di inizio.

Memoization nei linguaggi di programmazione

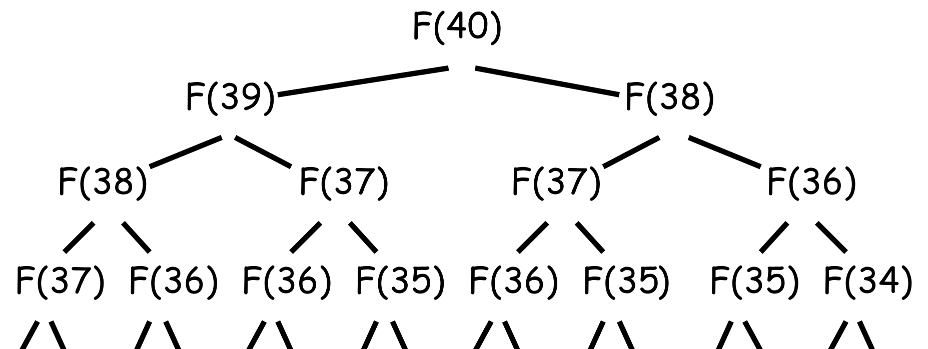
- Automatica. Molti linguaggi di programmazione funzionale, quali il Lisp, prevedono un meccanismo per rendere automatica la memoization

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

Lisp (efficiente)

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

Java (esponenziale



Interval Scheduling Pesato: Trovare una soluzione

- Domanda. Gli algoritmi di programmazione dinamica computano il valore ottimo. E se volessimo trovare la soluzione ottima e non solo il suo valore?
- Risposta. Facciamo del post-processing (computazione a posteriori).

```
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)
}
```

- **# chiamate ricorsive ≤ n ⇒ O(n).**

Interval Scheduling Pesato: Bottom-Up

- Programmazione dinamica bottom-up
- Per capire il comportamento dell'algoritmo di programmazione dinamica e` di aiuto formulare una versione iterativa dell'algoritmo.

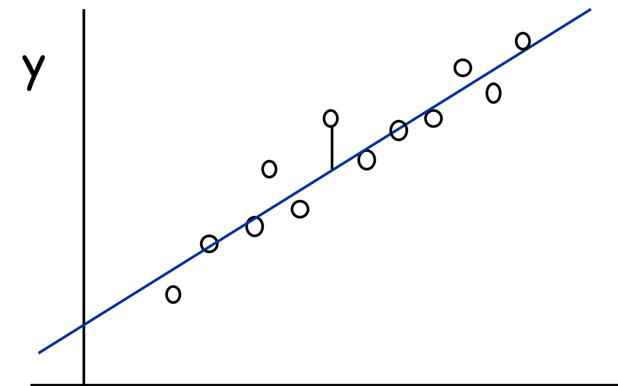
```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute 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])
}
```

- Correttezza: Con l'induzione su j si puo` dimostrare che ogni entrata M[j] contiene il valore OPT(j)
- Tempo di esecuzione: n iterazioni del for, ognuna della quali richiede tempo O(1) → tempo totale O(n)

Segmented Least Squares

- Minimi quadrati.
  - Problema fondazionale in statistica e calcolo numerico.
  - Dato un insieme P di n punti del piano $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
  - Trovare una linea L di equazione y = ax + b che minimizza la somma degli errori quadratici.

$$Error(L,P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

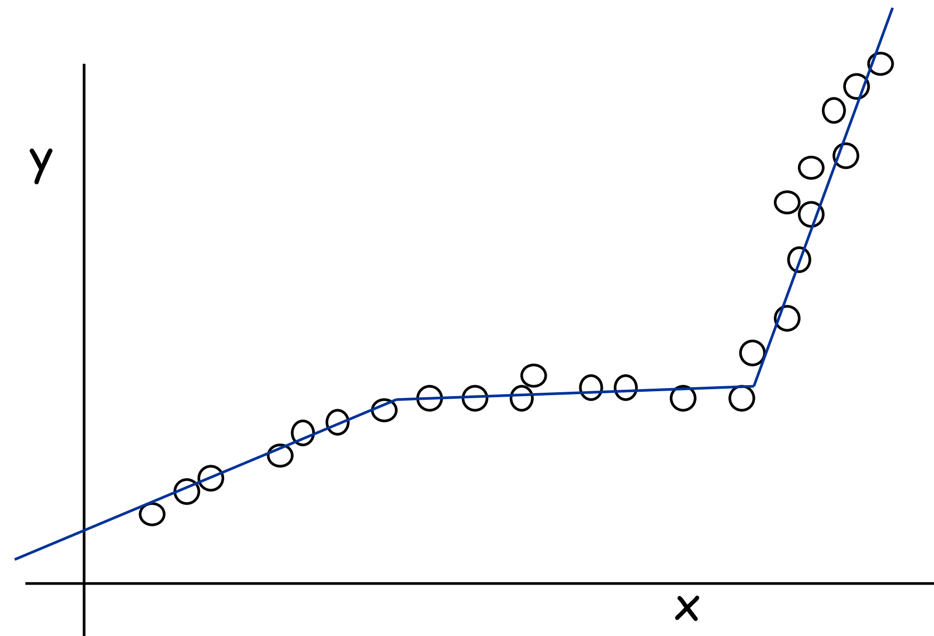


- Soluzione. Analisi ⇒ il minimo errore si ottiene usando la linea di equazione y = ax + b con a e b dati da

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

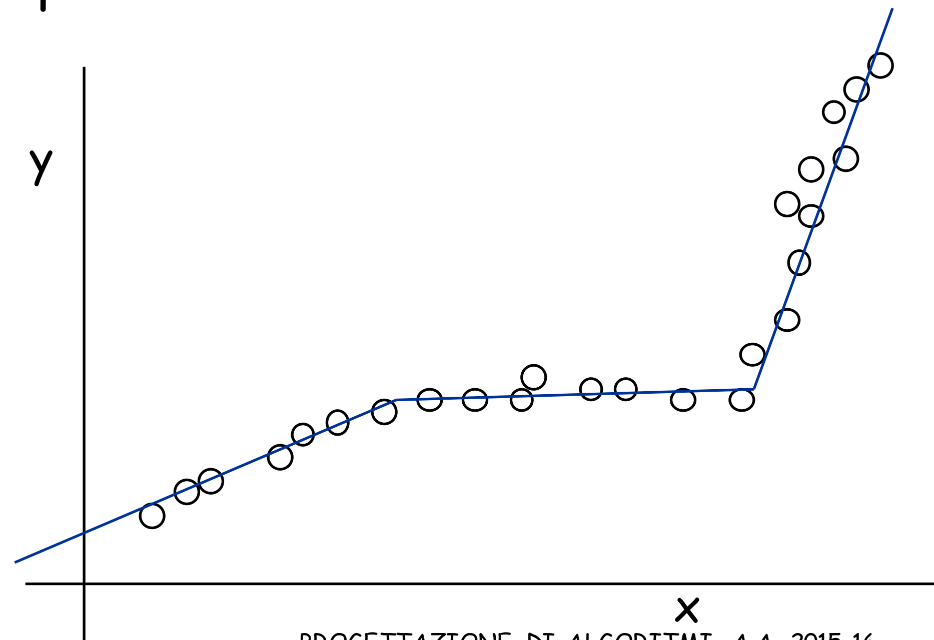
Segmented Least Squares

- **Esempio:** i punti in figura non possono essere ben approssimati usando un'unica linea. Se pero` usiamo tre linee riusciamo a ridurre di molto l'errore.



Segmented Least Squares

- Segmented least squares.
  - In generale per ridurre l'errore avremo bisogno di una sequenza di linee intorno alle quali si distribuiscono sottoinsiemi di punti di P.
  - Ovviamente se ci fosse concesso di usare un numero arbitrariamente grande di segmenti potremmo ridurre a zero l'errore:
    - Potremmo usare una linea ogni coppia di punti consecutivi.
- Domanda. Qual e` la misura da ottimizzare se vogliamo trovare un giusto compromesso tra accuratezza della soluzione e parsimonia nel numero di linee usate?



- Formulazione del problema Segmented Least Squares.
  - Dato un insieme P di n punti nel piano $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ con $x_1 < x_2 < \ldots < x_n$, vogliamo partizionare P in un certo numero m di sottoinsiemi $P_1, P_2, \ldots, P_m$ in modo tale che
  - Ciascun P_i e` costituito da punti contigui lungo l'asse delle ascisse
    - P_i viene chiamato segmento
  - La sequenza di linee $L_1, L_2, \ldots, L_m$ ottime rispettivamente per $P_1, P_2, \ldots, P_m$ minimizzi la somma delle 2 seguenti quantita
    - La somma delle somme degli errori quadratici

$$E = Error(L_1, L_2, \ldots, L_m; P_1, P_2, \ldots, P_m) = \sum_{j=1}^{m} \sum_{(x_i, y_i) \in P_j} (y_i - a_j x_i - b_j)^2$$

    - Il numero m di linee (pesato per una certa costante)

  - Penalita` : $E + C\,m$, per una certa costante $C > 0$.

Segmented Least Squares

- Il numero di partizioni in segmenti dei punti in P e` esponenziale
- La programmazione dinamica ci permette di progettare un algoritmo efficiente per trovare una partizione di penalita` minima
- A differenza del problema dell'Interval Scheduling Pesato in cui utilizzavamo una ricorrenza basata su due possibili scelte, per questo problema utilizzeremo una ricorrenza basata su un numero polinomiale di scelte.

Approccio basato sulla programmazione dinamica

- Notazione
  - $OPT(j)$ = costo minimo della penalita` per i punti $p_1, p_2, \ldots, p_j$.
  - $e(i, j)$ = minima somma degli errori quadratici per i punti $p_i, p_{i+1}, \ldots, p_j$.

- Per computare $OPT(j)$, osserviamo che
  - se l'ultimo segmento nella partizione di $\{p_1, p_2, \ldots, p_j\}$ e` costituito dai punti $p_i, p_{i+1}, \ldots, p_j$ per un certo i, allora
  - penalita` = $e(i, j) + C + OPT(i-1)$.

  - Il valore della penalita` cambia in base alla scelta di i
  - Il valore $OPT(j)$ e` ottenuto in corrispondenza dell'indice i che minimizza $e(i, j) + C + OPT(i-1)$

$$OPT(j) = \begin{cases} 0 & \text{se } j = 0 \\ \min\limits_{1 \le i \le j} \left\{ e(i,j) + C + OPT(i-1) \right\} & \text{altrimenti} \end{cases}$$

Segmented Least Squares: Algorithm

```
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]
}
```

Tempo di esecuzione. $O(n^3)$.

- Collo di bottiglia = dobbiamo computare il valore $e(i, j)$ per $O(n^2)$ coppie i, j. Usando la formula per computare la minima somma degli errori quadratici, ciascun e(i,j) e` computato in tempo O(n)

Algoritmo che produce la partizione

- Tempo di esecuzione $O(n^2)$ se abbiamo memorizzato i valori $e_{i,j}$

```
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, ..., p_j} and the result of
           Find-Segments(i − 1)
  Endif
```

Subset sums

- Input
  - n job 1,2,…,n
    - il job i richiede tempo $w_i > 0$.
  - Un limite W al tempo per il quale il processore puo` essere utilizzato
- Obiettivo: selezionare un sottoinsieme S degli n job tale che
- $\sum_{i \in S} w_i$ sia quanto piu` grande e` possibile, con il vincolo $\sum_{i \in S} w_i \leq W$

Greedy 1: ad ogni passo inserisce in S il job con peso piu` alto in modo che la duranta complessiva dei job in S non superi W

Esempio: Input una volta ordinato [W/2+1,W/2,W/2]. L'algoritmo greedy seleziona solo il primo mentre la soluzione ottima e` formata dagli ultimi due.

Greedy 2: ad ogni passo inserisce in S il job con peso piu` basso in modo che la duranta complessiva dei job in S non superi W

Esempio: Input [1,W/2,W/2] una volta ordinato . L'algoritmo greedy seleziona i primi due per un peso complessibo di 1+w/2. mentre la soluzione ottima e` formata dagli ultimi due di peso complessivo w.

Programmazione dinamica: falsa partenza

- **Def.** OPT(i) = valore della soluzione ottima per {1, …, i}.

  - Caso 1: OPT non seleziona i.
    - OPT seleziona la soluzione ottima per { 1, 2, …, i-1 }

  - Caso 2: OPT seleziona i.
    - Prendere i non implica immediatamente l'esclusione di altri elementi.
    - Se non conosciamo i job selezionati prima di i, non sappiamo neanche se c'e` tempo sufficiente per eseguire i

- **Conclusione.** Approccio sbagliato!

Programmazione dinamica: approccio corretto

- Per esprimere il valore della soluzione ottima per un certo i in termini dei valori delle soluzioni ottime per input piu` piccoli di i, dobbiamo introdurre un limite al tempo totale da dedicare all'esecuzione dei job selezionati prima di i.

- Per ciascun j, consideriamo il valore della soluzione ottima per i job 1, …, j con il vincolo che il tempo necessario per eseguire i job selezionati non superari un certo w.

-

- Def. OPT(i, w) = valore della soluzione ottima per i job 1, …, i con limite w sul tempo di utilizzo del processore.

Programmazione dinamica: approccio corretto

- Def. OPT(i, w) = valore della soluzione ottima per i job 1, …, i con limite w sul tempo di utilizzo del processore.

- Caso 1: OPT non seleziona il job i.
  - OPT produce la soluzione ottima per { 1, 2, …, i-1 } in modo che il tempo di esecuzione totale dei job non superi w

- Case 2: OPT seleziona il job i.
  - OPT produce la soluzione ottima per { 1, 2, …, i-1 } in modo che il tempo di esecuzione totale dei job non superi $w - w_i$

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\left\{ OPT(i-1, w),\quad w_i + OPT(i-1, w - w_i) \right\} & \text{otherwise} \end{cases}$$

Subset sums: algoritmo

- Versione iterativa in cui si computa la soluzione in modo bottom-up
- Si riempie un array bidimensionale nxW a partire dalle locazioni di indice di riga i piu` piccolo

```
Input: n, w1,…,wN,W

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

for i = 1 to n
   for w = 0 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}

return M[n, W]
```

Subset sums: esempio di esecuzione dell'algoritmo

Limite $W = 6$, durate job $w_1 = 2, w_2 = 2, w_3 = 3$

i \ w	0	1	2	3	4	5	6
3							
2							
1							
0	0	0	0	0	0	0	0

Initial values

i \ w	0	1	2	3	4	5	6
3							
2							
(1)	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0

Filling in values for $i = 1$

i \ w	0	1	2	3	4	5	6
3							
(2)	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0

Filling in values for $i = 2$

i \ w	0	1	2	3	4	5	6
(3)	0	0	2	3	4	5	5
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0

Filling in values for $i = 3$

Subset sums: correttezza algoritmo

- Induzione sui i. Dimostriamo che all'i-esima iterazione ogni riga di M con indice j compreso tra 0 e i contiene i valori OPT(j,0),OPT(j,1),….,OPT(j, W)

- Base induzione. i=0: La riga 0 ha correttamente tutte le entrate uguali a 0
- Passo induttivo: Supponiamo che alla iterazione (i-1)≥0, le righe di indice j compreso tra 0 e i-1 contengano i valori OPT(j,0),OPT(j,1),….,OPT(j, W).
- Vediamo cosa succede all'i-esima iterazione.
- Per ipotesi induttiva M[i-1, w]=OPT(i-1,w) ed M[i-1, w-w_i]=OPT(i-1,w-w_i)
- Per cui all'i-esima iterazione, l'algoritmo pone

M[i, w] =OPT(i-1,w) **se w_i > w**

M[i, w] = max {OPT[i-1, w], w_i + OPT[i-1, w-w_i]} altrimenti

- Per cui M[i,w]=OPT(i,w)
-

Subset sums: tempo di esecuzione algoritmo

- Tempo di esecuzione. $\Theta(n\ W)$.
  - Non e` polinomiale nella dimensione dell'input! !
  - "Pseudo-polinomiale": L'algoritmo e` efficiente quando W ha un valore ragionevolmente piccolo.

  - Se volessimo produrre la soluzione ottima, potremmo scrivere un algoritmo simile a quelli visti prima in cui la soluzione ottima si ricostruisce andando a ritroso nella matrice M. Tempo O(n).

Problema dello zaino

- Input
  - n oggetti ed uno zaino
  - L'oggetto i pesa $w_i > 0$ chili e ha valore $v_i > 0$.
  - Lo zaino puo` trasportare fino a W chili.
- Obiettivo: riempire lo zaino in modo da massimizzare il valore totale degli oggetti inseriti.
- Esempio: { 3, 4 } ha valore 40.

W = 11

Oggetto	Valore	Peso
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

- Greedy: seleziona ad ogni passo l'oggetto con il rapporto v_i/w_i piu` grande in modo che il peso totale dei pesi selezionati non superi w
- Esempio: { 5, 2, 1 } ha valore = 35 $\Rightarrow$ greedy non e` ottimo

Problema dello zaino

- Input
  - n oggetti ed uno zaino
  - L'oggetto i pesa w_i > 0 chili e ha valore v_i > 0.
  - Lo zaino puo` trasportare fino a W chili.
- Obiettivo: riempire lo zaino in modo da massimizzare il valore totale degli oggetti inseriti.
- Corrisponde al problema subset sums quanto $v_i = w_i$ per ogni i.

Problema dello zaino: estensione approccio usato per Subset Sums

- **Def.** OPT(i, w) = valore della soluzione ottima per gli oggetti 1, …, i con limite di peso totale w.

- Caso 1: OPT non seleziona l'elemento i.
  - OPT produce la soluzione ottima per { 1, 2, …, i-1 } in modo che il peso totale dgli elementi non superi w

- Case 2: OPT seleziona l' elemento i.
  - OPT produce la soluzione ottima per { 1, 2, …, i–1 } in modo che il peso totale degli elementi non superi w– w_i

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

Problema dello zaino: algoritmo

```
Input: n, W,w1,…,wN, v1,…,vN

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

for i = 1 to n
   for w = 0 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]
```

Algoritmo per il problema della zaino: esempio

W →

n ↓

	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 }
value = 22 + 18 = 40

W = 11

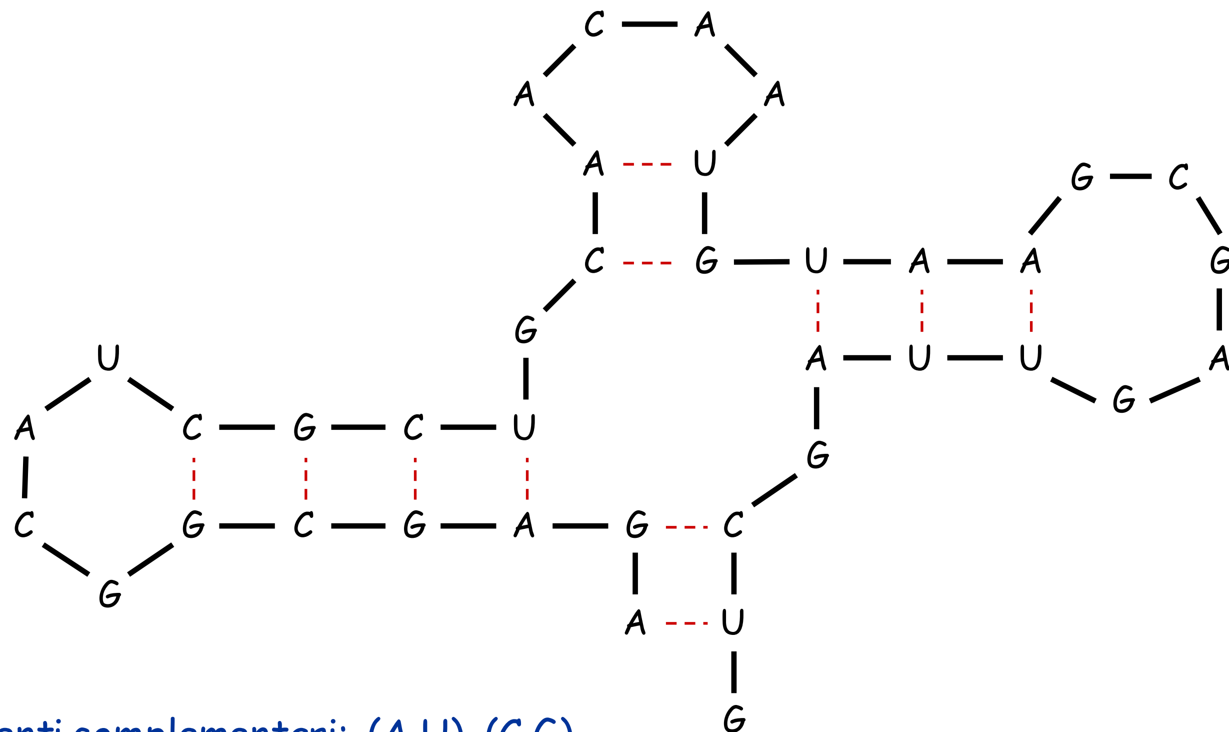
Oggetto	Valore	Peso
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

RNA Secondary Structure

- L'RNA e` una catena di nucleotidi spesso costituita da un singolo filamento che tende a ripiegarsi su se' stesso formando coppie di basi. Questa struttura, chiamata struttura secondaria e` essenziale per capire il comportamento delle molecole.

RNA. Stringa B = $b_1b_2\ldots b_n$ sull'alfabeto di simboli (basi) { A, C, G, U }.

Esempio: **GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA**



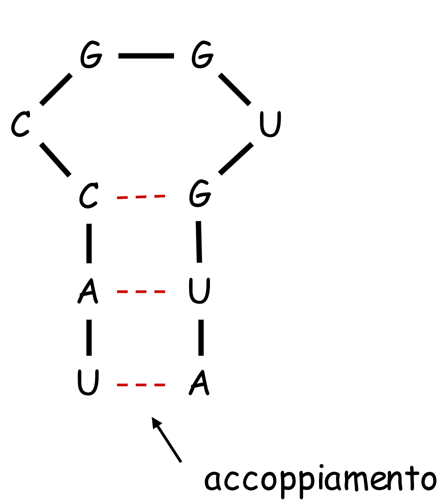
Accoppiamenti complementari: (A,U), (C,G)

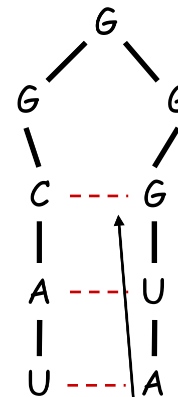
La struttura secondaria dell'RNA

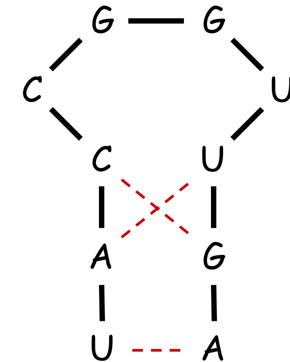
- Sia $B=b_1,\ldots,b_n$ una molecola di RNA a filamento singolo
- La struttura secondaria e` formata da un insieme di coppie $S = \{ (b_i, b_j) \}$: $i,j \in \{1,\ldots,n\}\}$ che soddisfano le seguenti proprieta`
  - S e` un matching: ogni $b_i$ e` accoppiato con un solo b_j e viceversa
  - Accoppiamento complementare: le coppie in S possono essere solo della forma: (A,U), (U,A), (C,G), o (G,C).
  - Nessuna curva a gomito: Tra le estremita` dell'arco si interpongono almeno 4 basi. Se $(b_i, b_j) \in S$, allora $i < j - 4$.
  - Condizione di non incrocio: Se (b_i, b_j) e (b_k, b_l) sono due coppie in S allora non puo` verificarsi che $i < k < j < l$.

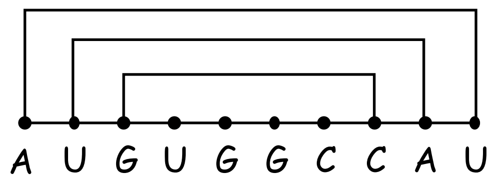
Struttura secondaria dell'RNA

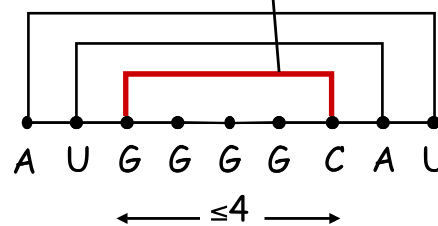
- Esempi.

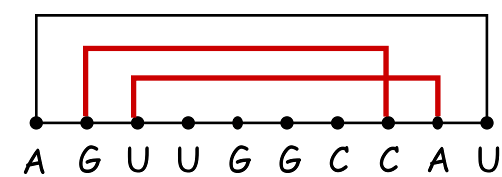










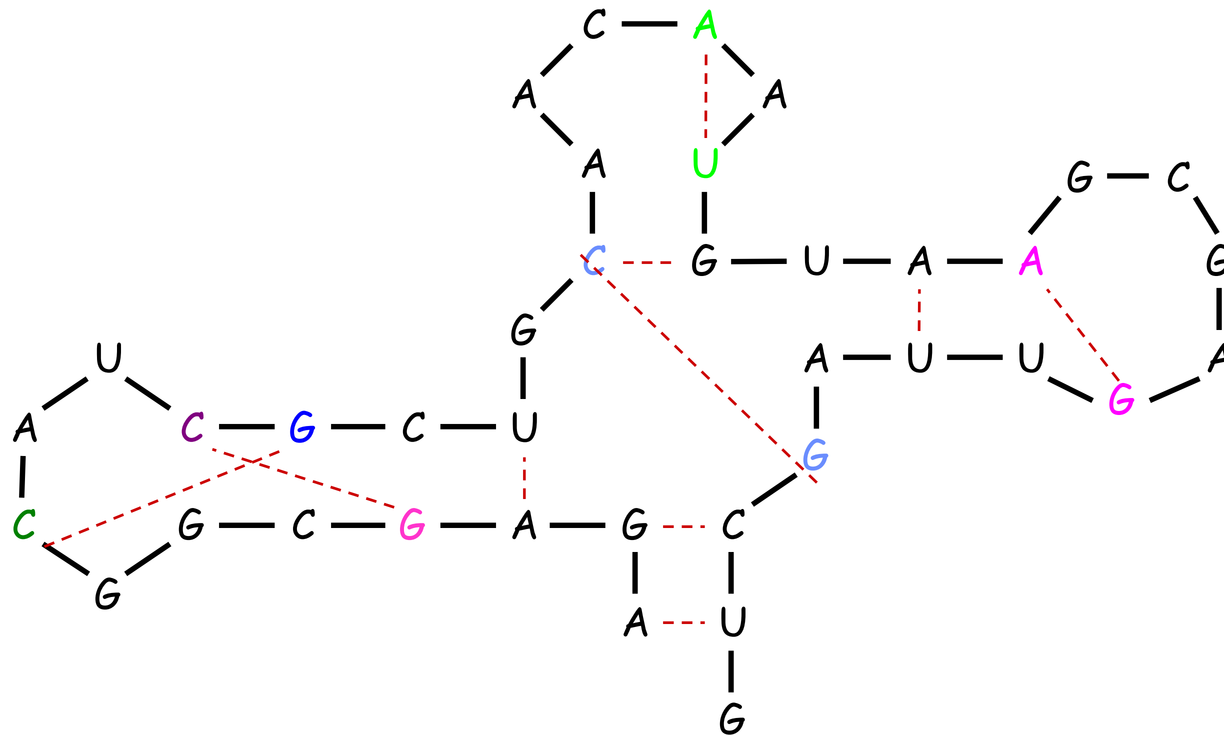


ok

Curva a gomito

incrocio

GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA



Non e` possibile avere la coppia (G,C) perche` altrimenti C comparirebbe in due coppie

Non e` possibile avere le due coppie (G, C) e (C,G) perche` le coppie si incrociano : G precede il C che precede C che precede G

Non e` possibile avere la coppia (G,A) perche` gli elementi non sono complementari

Non e` possibile avere la coppia (U,A) perche` gli elementi sono troppo vicini nella sequenza

La struttura secondaria dell'RNA

Problema della predizione della struttura secondaria: L'ipotesi usata comunemente e` che la molecola di RNA crei la struttura secondaria usando l'energia libera totale ottimale. Qui assumiamo che l'energia libera totale sia uguale al numero di accoppiamenti.

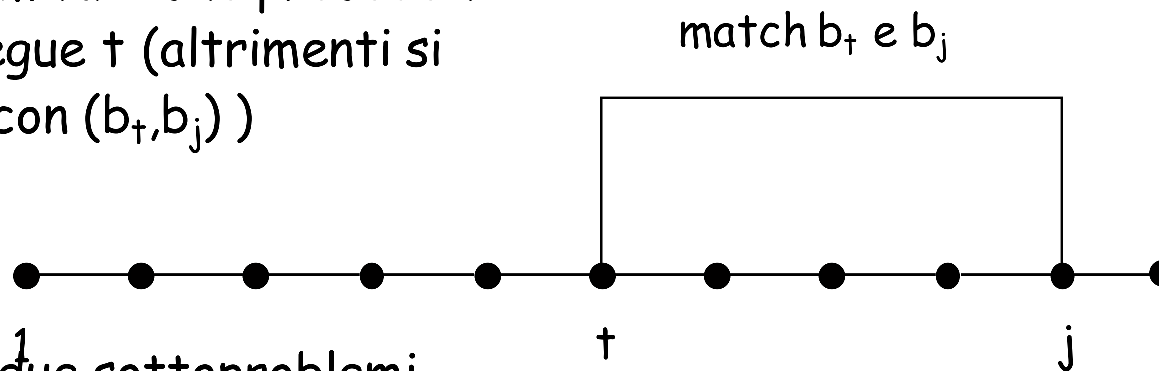
- Obiettivo. Data una molecola di RNA $B = b_1b_2 \ldots b_n$, trovare una struttura secondaria S che minimizza l'energia libera totale. Nella nostra ipotesi semplificativa, cio` equivale a trovare una struttura secondaria S che massimizza il numero di accoppiamenti.

Programmazione dinamica: approccio sbagliato

- OPT(j) = massimo numero di coppie nella sottostringa $b_1 b_2 \ldots b_j$.
- OPT(j)=0 per $j \leq 5$ (impossibile non avere curve a gomito)
- Caso 1: b_j non e` accoppiato: In questo caso OPT(j)=OPT(j-1)

Caso 2: b_j e` accoppiato. Sia $b_t$ la base con cui e` accoppiato.

La condizione di non incrocio implica che nessuna coppia puo` avere un'estremita` che precede t e l'altra che segue t (altrimenti si incrocerebbe con (b_t, b_j))



- Risulta in due sottoproblemi.
  - Trovare la struttura secondaria in: $b_1 b_2 \ldots b_{t-1}$.
  - Trovare la struttura secondaria in: $b_{t+1} b_{t+2} \ldots b_{j-1}$. NON BASTA SPECIFICARE UN SOLO INDICE !

Programmazione dinamica: approccio corretto

- Abbiamo bisogno di specificare un intervallo $(i,j)=\{b_i,\ldots,b_j\}$ per $i<j$
- Notazione. $OPT(i, j)$ = numero massimo di coppie in una struttura secondaria di $b_i b_{i+1} \ldots b_j$.

NB: Se $i \geq j - 4$ allora $OPT(i, j) = 0$ per la condizione sull'assenza di curve strette
Per convenienza notazionale definiamo $OPT(i,j)$ anche per $i>j$ e lo poniamo uguale a 0.

- Caso 1. La base b_j non e` accoppiata.
  - $OPT(i, j) = OPT(i, j-1)$
- Caso 2. La base b_j e` accoppiata. Sia $b_t$ la base con cui e` accoppiata, con t tale che $i \leq t < j - 4$.
  - La condizione di non incrocio implica che nessuna coppia puo` avere un'estremita` in $(i,t-1)$ e l'altra in $(t+1,j-1)$ (altrimenti si incrocerebbe con (b_t,b_j))
  - Il problema quindi consiste nel risolvere i 2 sottoproblemi $OPT(i, t-1)$ e $OPT(t+1, j-1)$
  - $OPT(i, j) = 1 + \max_t \{ OPT(i, t-1) + OPT(t+1, j-1) \}$. Questo massimo e` computato considerando solo i valori di t per i quali puo` esistere la coppia (b_t,b_j) in base alle regole dell'accoppiamento complementare e dell'assenza di curve strette.

  $OPT(i, j)$ e` quindi il max tra i valori ottimi del caso 1 e quello del caso 2

Programmazione dinamica sugli intervalli

- L'algoritmo computa le soluzioni ottimali dei sottoproblemi in maniera bottom up.
- Calcola prima la soluzione per gli intervalli piu` piccoli e poi per quelli piu` grandi.

```
RNA(b1,…,bn) {
   for k = 5, 6, …, n-1
     for i = 1, 2, …, n-k
       j = i + k
       M[i, j]=max{M[i, j-1],1 + maxt { M[i, t-1] + M[t+1, j-1] }}

   return M[1, n]
}
```

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

- Tempo di esecuzione. $O(n^2)$ iterazioni del corpo dei due for innestati. Ciascuna iterazione richiede $O(j-i)=O(n)$ → Tempo algoritmo $O(n^3)$.

Esempio di esecuzione dell'algoritmo per la predizione della struttura secondaria

Sono riportate solo le entrate M[i,j] per i valori di i e j t.c. j-i>4

RNA sequence *ACCGGUAGU*

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

Initial values

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

Filling in the values for $k = 5$

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

Filling in the values for $k = 6$

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

Filling in the values for $k = 7$

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

Filling in the values for $k = 8$

Esercizio 27 cap. 6

- I proprietari di una pompa di carburante devono confrontarsi con la seguente situazione:
- Hanno un grande serbatioio che immagazzina gas; il serbatoio puo` immagazzinare fino ad L galloni alla volta.
- Ordinare carburante e` molto costoso e per questo essi vogliono farlo raramente. Per ciascun ordine pagano un prezzo fisso P in aggiunta al costo della carburante.
- Immagazzinare un gallone di carburante per un giorno in piu` costa c dollari per cui ordinare carburante troppo in anticipo aumenta i costi di immagazzinamento.
- Essi stanno progettando di chiudere per una settimana durante l'inverno e vogliono che per allora il serbatoio sia vuoto.
- In base all'esperienza degli anni precedenti, essi sanno esattamente di quanto carburante hanno bisogno. Assumendo che chiuderanno dopo n giorni e che hanno bisogno di g_i galloni per ciascun giorno i=1,…n e che al giorno 0 il serbatoio e` vuoto, dare un algoritmo per decidere in quali giorni devono effettuare gli ordini e la quantita` di carburante che devono ordinare in modo da minimizzare il costo.

Esercizio 27 cap. 6: Soluzione

- Supponiamo che il giorno 1 i proprietari ordinino carburante per i primi i-1 giorni: $g_1+g_2+\dots+g_{i-1}$
- Il costo di questa operazione si traduce in un costo fisso di P piu` un costo di $c(g_2+2g_3+3g_4\dots+(i-2)g_{i-1})$

- Sia OPT(d) il costo della soluzione ottima per il periodo che va dal giorno d al giorno n partendo con il serbatoio vuoto
- La soluzione ottima da d ad n include sicuramente un ordine al giorno d per un certo quantitativo di carburante. Sia f il giorno in cui verra` fatto il prossimo ordine.
  - La quantita ordinata il giorno d deve essere $g_d+g_{d+1}+\dots+g_{f-1}$. $(g_d+g_{d+1}+\dots+g_{f-1} \le L)$
  - Il costo connesso a questo ordine e` $P+c(g_{d+1}+2g_{d+2}+3g_{d+3}\dots+(f-1-d)g_{f-1})$
  - Il costo della soluzione ottima da d ad n se il secondo ordine in questo intervallo avviene al tempo f e` $P+\sum_{i=d}^{f-1} c(i-d)g_i + OPT(f)$

$$OPT(d) = P + \min_{f>d:\sum_{i=d}^{f-1} g_i \le L} \sum_{i=d}^{f-1} c(i-d)g_i + OPT(f)$$

Esercizio 27 cap. 6: Soluzione

Possiamo scrivere un algoritmo iterativo che computa le soluzioni a partire da d = n fino a d=1 e memorizza le soluzioni in un array

```
Input: n, L, g1,…,gn
A[d,f]  //contiene la somma dei gi per i=d,…,f
S[d,f]   //contiene la somma dei gi(d-i)per i=d,…f
M[d] //contiene soluzione ottima da d ad n


For d = 1 to n
  A[d,d]=gd
  S[d,d]=0

For d = n to 1{
   min= large_value
   For  x=d to n  {      //x=f-1
      If A[d,x-1]+gx<=L{
         A[d,x]=A[d,x-1]+gx
         S[d,x] =S[d,x-1]+ (x-d)gx
         If P+S[d,x]+ M[x+1]<min
           min= P+S[d,x]+ M[x+1]
         M[d]=min}
   Else  break }}//ha computato l'ottimo per giorni da d ad n
return M[1]
```

$O(n^2)$

Allineamento di sequenze

- Quanto sono simili le due stringhe seguenti ?
  - **ocurrance**
  - **occurrence**
  - Allineamo i caratteri
  - Ci sono diversi modi per fare questo allineamento
  - Qual e` migliore?

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

6 mismatch, 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 mismatch, 3 gap

Edit Distance

- Applicazioni.
  - Base per il comando Unix diff.
  - Riconoscimento del linguaggio.
  - Biologia computazionale.

- Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]
  - Gap penalty δ;
  - Mismatch penalty α_{pq}. Si assume $\alpha_{pp}=0$
  - Costo = somma delle due penalita`

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

$2\delta + \alpha_{CA}$

Applicazione del problema dell'allineamento di sequenze

- I problemi su stringhe sorgono naturalmente in biologia: il genoma di un organismo e` suddiviso in molecole di DNA chiamate cromosomi, ciascuno dei quali serve come dispositivo di immagazzinamento chimico.
- Di fatto, si puo` pensare ad esso come ad un enorme nastro contenente una stringa sull'alfabeto {*A*,*C*,*G*,*T*}. La stringa di simboli codifica le istruzioni per costruire molecole di proteine: usando un meccanismo chimico per leggere porzioni di cromosomi, una cellula puo` costruire proteine che controllano il suo metabolismo.
-

Continua nella prossima slide

Applicazione del problema dell'allineamento di sequenze

- Perche` le somiglianze tra stringhe sono rilevanti in questo scenario?
- Le sequenze di simboli nel genoma di un organismo determinano le proprieta` dell'organismo.
- Esempio. Supponiamo di avere due ceppi di batteri X e Y che sono strettamente connessi dal punto di vista evolutivo.
- Supponiamo di aver determinato che una certa sottostringa nel DNA di X sia la codifica di una certa tossina.
- Se scopriamo una sottostringa molto simile nel DNA di Y, possiamo ipotizzare che questa porzione del DNA di Y codifichi un tipo di tossina molto simile a quella codificata nel DNA di X.
- Esperimenti possono quindi essere effettuati per convalidare questa ipotesi.
- Questo e` un tipico esempio di come la computazione venga usata in biologia computazionale per prendere decisioni circa gli esperimenti biologici.

Allineamento di sequenze

- Abbiamo bisogno di un modo per allineare i caratteri di una stringa
- Def. insieme di coppie e` un matching se ogni elemento appartiene ad una sola coppia
- Def. Le coppie (x_i, y_j) e $(x_{i'}, y_{j'})$ si incrociano se $i < i'$ ma $j > j'$.

Def. Un allineamento M e` un insieme di coppie (x_i, y_j) tali che

- M e` un matching
- M non contiene coppie che si incrociano

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

Allineamento di sequenze

- Obiettivo: Date due stringhe $X = x_1 x_2 \ldots x_m$ e $Y = y_1 y_2 \ldots y_n$ trova l'allineamento di minimo costo.

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

- Affermazione.
- Dato un allineamento M di due stringhe $X = x_1 x_2 \ldots x_m$ e $Y = y_1 y_2 \ldots y_n$, se in M non c'e` la coppia $(x_m,y_n)$ allora o $x_m$ non e` accoppiato in M o y_n non e` accoppiato in M.
- Dim. Supponiamo che x_m e y_n sono entrambi accoppiati, x_m con y_j e y_n con x_i. In altre parole M contiene le coppie (x_m,y_j) e (x_i,y_n). Siccome $i<m$ ma $n>j$ allora si ha un incrocio e cio` contraddice il fatto che M e` allineamento.

Allineamento di sequenze: struttura del problema

- Def. OPT(i, j) = costo minimo dell'allineamento delle due stringhe $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$.
  - Caso 1: OPT accoppia x_i e y_j.
    - OPT(i, j) = Costo dell'eventuale mismatch tra x_i e y_j + costo minimo dell'allineamento di $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_{j-1}$
  - Caso 2a: OPT lascia x_i non accoppiato.
    - OPT(i, j) = Costo del gap x_i + costo minimo dell'allineamento di $x_1 x_2 \ldots x_{i-1}$ e $y_1 y_2 \ldots y_j$
  - Case 2b: OPT lascia y_j non accoppiato.
    - OPT(i, j) = Costo del gap y_j + costo minimo dell'allineamento di $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_{j-1}$

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

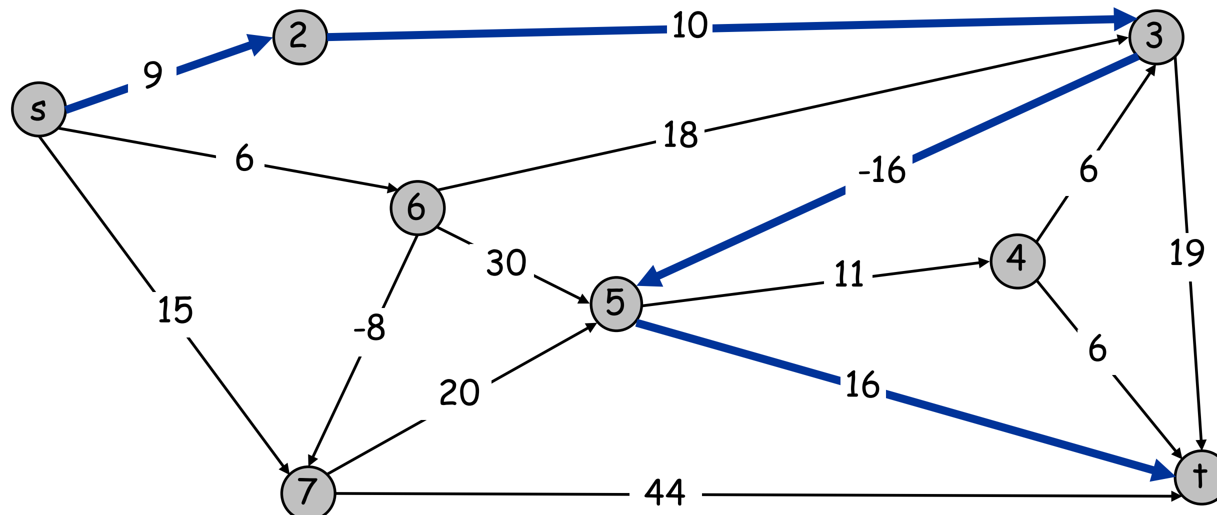
Allineamento di sequenze: algoritmo

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α) {
   for i = 0 to m
      M[i, 0] = iδ
   for j = 0 to n
      M[0, j] = 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. Tempo e spazio $\Theta(mn)$.
- Parole inglesi: $m, n \leq 10$.
- Applicazioni di biologia computazionale: $m = n = 100{,}000$.
- Quindi mxn=10 miliardi . OK per il tempo ma non per lo spazio (10GB)

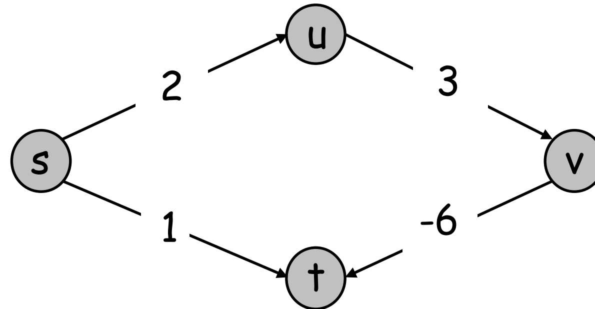
- **Problema del percorso piu` corto.** Dato un grafo direzionato G = (V, E), con pesi degli archi $c_{vw}$, trovare il percorso piu` corto da s a t.

- **Esempio.** I nodi rappresentano agenti finanziari e c_{vw} e` il costo di una transazione che consiste nel comprare dall'agente v e vendere immediatamente a w.

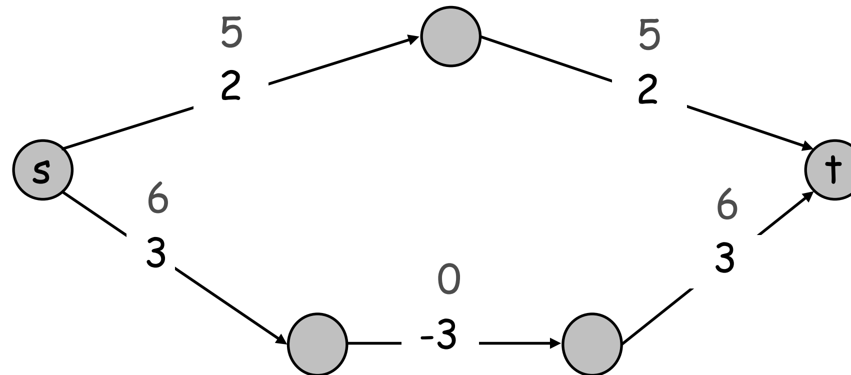


Cammini minimi in presenza di archi con costo negativo

- Dijkstra. Puo` fallire se ci sono archi di costo negativo

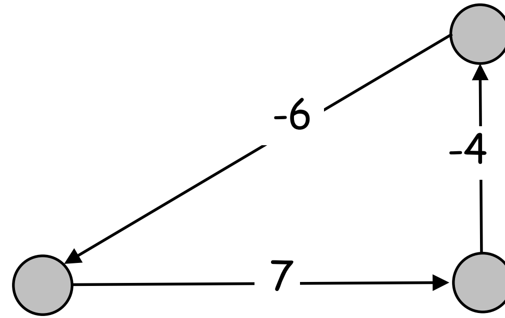


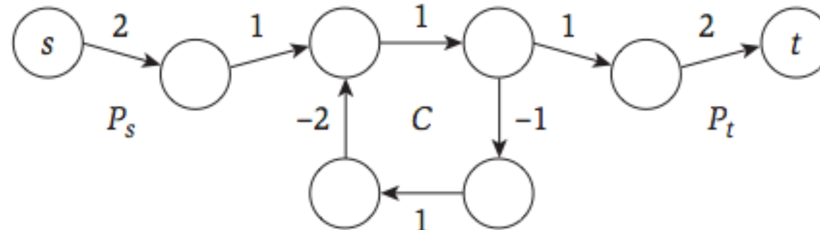
- Re-weighting. Aggiungere una costante positiva ai pesi degli archi potrebbe non funzionare.



Cammini minimi in presenza di archi con costo negativo

- Ciclo di costo negativo.

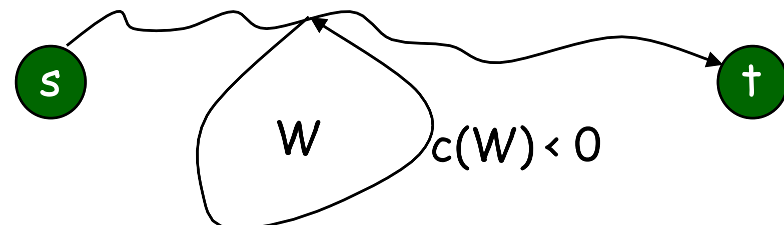




Cammini minimi in presenza di archi con costo negativo

Osservazione. Se qualche percorso da s a t contiene un ciclo di costo negativo allora non esiste un percorso minimo da s a t. In caso contrario esiste un percorso minimo da s a t che e` semplice (nessun nodo compare due volte sul percorso).

- Dim. Sia P un percorso da s a t con un ciclo C di costo negativo –c. Ogni volta che attraversiamo il ciclo riduciamo il costo del percorso di un valore pari a c. Cio` rende impossibile definire il costo del percorso minimo perche` dato un percorso riusciamo sempre a trovarne uno di costo minore attraversando il ciclo C .
  Sia ora P un percorso da s a t privo di cicli di costo negativo. Supponiamo che un certo vertice v appaia almeno due volte in P. C'e` quindi in P un ciclo che contiene v e che per l' ipotesi deve avere costo positivo. In questo caso potremmo rimuovere le porzioni di P tra due occorrenze consecutive di v in P senza far aumentare il costo del percorso.



Cammini minimi: Programmazione dinamica

- **Def.** OPT(i, v) = lunghezza del cammino piu` corto P per andare da v a t che consiste di al piu` i archi

  - Caso 1: P usa al piu` i-1 archi.
    - OPT(i, v) = OPT(i-1, v)

  - Caso 2: P usa esattamente i archi.
    - se (v, w) e` il primo arco allora P e` formato da (v, w) e dal percorso piu` corto da w a t di al piu` i-1 archi

$$OPT(i,v) = \min_{(v,w)\in E} \{ OPT(i-1,\ w) + c_{vw} \}$$

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

Cammini minimi: Programmazione dinamica

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

Dove si usa l'osservazione di prima sul fatto che in assenza di cicli negativi il percorso minimo e` semplice?

Ecco dove….

Osservazione. Se non ci sono cicli di costo negativo allora OPT(n-1, v) = lunghezza del percorso piu` corto da v a t.

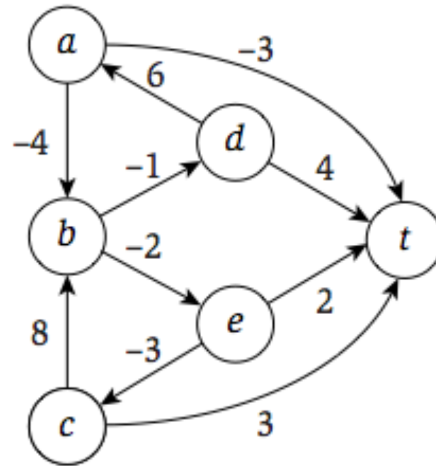
Dim. Dall'osservazione precedente se non ci sono cicli negativi allora il percorso piu` corto da v a t e` semplice e di conseguenza contiene al piu` n-1 archi

Algoritmo di Bellman-Ford per i cammini minimi

```
Shortest-Path(G, t) {
   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 $\Theta(n^2)$.
- Trovare il percorso minimo. Memorizzare un successore per ogni entrata della matrice

Bellman-Ford: esempio



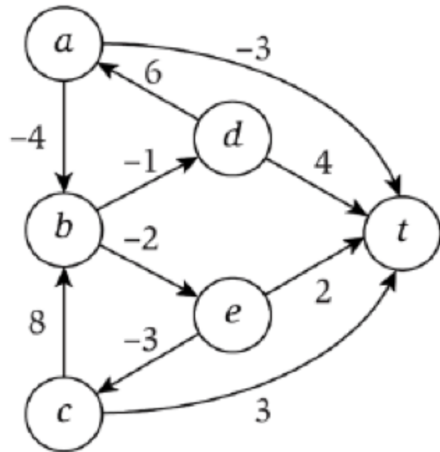
(a)

	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)

Bellman-Ford: esempio

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



	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

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))

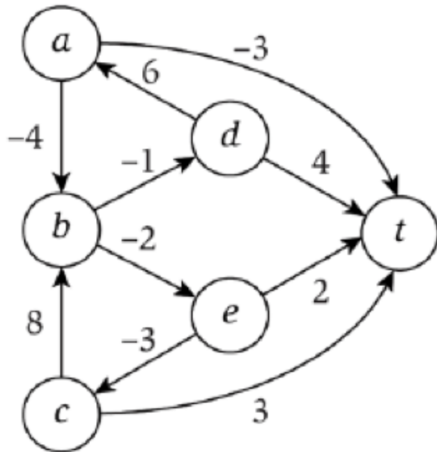
OPT(3,b) = min(OPT(2,b), min(OPT(2,e)+c_{be},OPT(2,d)+c_{bd}) b-e
= min(0, min(0-2,3-1))

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,c), 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

Bellman-Ford: esempio



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

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

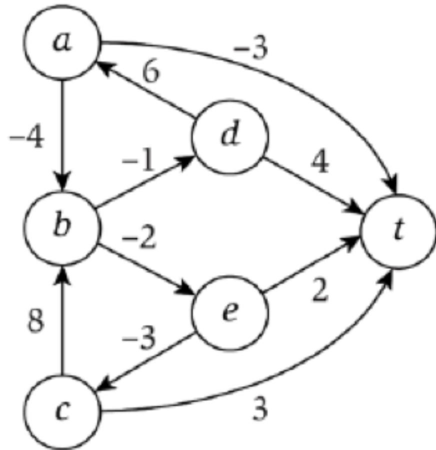
$OPT(1,c) = \min(OPT(0,c), \min(OPT(0,b)+c_{cb}, OPT(0,t)+c_{ct}\ )$ c-t
$= \min(\ \infty,\ \min(\infty+8, 0+3)\ )$

	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

a - b - e - c - t

-4 -2 -3 +3 = -6

Bellman-Ford: esempio



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

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

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

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

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

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

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